HIGH SPEED CONVOLUTION USING RESIDUE NUMBER SYSTEMS

by

KURT ANTHONY LOCHER

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE IN ELECTRICAL ENGINEERING

and

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1989

Signature of Author_____

Department of Electrical Engineering and Computer Science

January 21, 1989

Certified by_____

Bruce R. Musicus

Thesis Supervisor (Academic)

Certified by_____

Paul Hogan

Thesis Supervisor (Raytheon Corporation)

Accepted by_____

Arthur C. Smith

Chairman, Department Committee on Graduate Students

HIGH SPEED CONVOLUTION USING RESIDUE NUMBER SYSTEMS

by

KURT ANTHONY LOCHER

Submitted to the

Department of Electrical Engineering and Computer Science

on January 23, 1988 in partial fulfillment of the requirements

for the degrees of

Bachelor of Science in Electrical Engineering

and

Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis investigates architectures for high speed signal convolution using residue number system techniques. The focus is on VLSI implementations, a choice that gives two parameters by which judge different architectures: speed and hardware size. Although the focus is on the standard residue number system, the same concepts can be directly applied to quadratic residue number system architectures as well. Two general design classes are developed with several detailed implemetations investigated in each class.

Using the size and speed results of the detailed designs, a design aid was developed that prunes the design space leaving a core group of designs with optimal speed/size combinations.

One of the largest disadvantages of residue number system implementations of computational hardware is the large overhead associated with the conversion into and out of residue representation. To provide a comparision between residue number system architectures and conventional binary architectures a design is presented that uses several of the design optimizations developed for the residue designs.

Thesis Supervisor: Bruce R. Musicus

Title: Assistant Professor of Electrical Engineering

Acknowledgements

First, I would like to thank Bruce Musicus for all of the great ideas that flowed during our discussions on residue architectures. I'm sure that I would not have gotten as far if not for his inspiration.

I would also like to thank Frank Horrigan for introducing me to this subject and Paul Hogan for agreeing to supervise me on the Raytheon side.

Finally, to Claire who put up with me through the entire process, I'm finally finished...

Table of Contents

List of Abbreviations

CRT	Chinese Remainder Theorem
FIR	finite impulse response (filter)
IIR	infinite impulse response (filter)
MSI	medium scale integration
MUX	multiplexor
RNS	residue number system
SSI	small scale integration
VLSI	very large scale integration
XNOR	exclusive nor (gate)
XOR	exclusive or (gate)
WSI	wafer scale integration

List of Figures

Chapter 1 Introduction

Residue Number Systems (RNS) use a number theory concept of an alternate number representation, that mathematicians have been aware of since the 1800's, to add parallelism to certain computations. RNS was first investigated for application to digital computation in the early 1950's. A conclusive book on the subject was published in 1967 by Szabo and Tanaka, and since then the basic ideas have not advanced significantly. Recently there has been a renewed interest in RNS applications, not because of new theory, but instead as a result of advancing integrated circuit technology. Current VLSI and WSI technologies are ideally suited to RNS applications allowing high speed and circuit density.

Unfortunately, the advances in VLSI technology also open the possibility for other, non-RNS, designs to be applied. The conversion processes from standard binary to the RNS representation and from RNS to binary add a significant amount of overhead in both hardware and latency. As a result, the computation must be very large for the benefits of RNS to overcome the conversion overhead.

There are a few applications for which RNS is especially suited; the most common is the evaluation of a long convolution sum or equivalently an FIR filter. The convolution operation is used for both filtering and correlation which occur in radar, sonar, and communications applications. Frequently, the computation is performed using analog techniques because of size and speed limitations of digital circuitry. Maybe RNS can change this.

This thesis will focus on the RNS implementation of the FIR filter. The goal is to create a design aid that searches the possible architectures finding an optimum design for a given convolution problem. The user will be able to input the dynamic range of the filter coefficients and input and the length of the filter, and the design aid will return the optimum set of designs. Most of the effort is aimed toward creating a sufficiently rich set of implementations that provide some variety in speed and hardware size. The design aid itself is simple exercise in programming a search algorithm.

Chapter 2 FIR Filter Background

A finite impulse response (FIR) filter performs the discrete time convolution of an input signal with a fixed (finite length) system response. Mathematically, the convolution expression for a length N system response is written as follows:

$$y[n] = \sum_{i=0}^{N-1} h[i]x[n-i] \qquad (1)$$

This operation is commonly denoted shorthand by the expression y[n] = x[n] * h[n] where x[n] is the input sequence, h[n] is the system impulse response, and y[n] is the output sequence. This convolution becomes an extremely computation intensive operation for large N because N multiplies are needed to compute each output point.

Finite impulse response filters have a number of advantages over their close cousins, the infinite impulse response (IIR) filters[1]. First, FIR filters can be realized nonrecursively which guarantees stable operation. Although stability issues appear to be a filter design problem that could be solved on paper before implementation, a filter that is stable on paper can become unstable because of the limited dynamic range of finite register lengths or the coefficient truncation. Second, FIR filters can be designed to have linear phase. In the applications of radar and communications the frequency dispersion caused by the nonlinear phase of IIR filters can be harmful to the performance of the system. Finally, roundoff noise caused by finite register lengths can be minimized with an FIR structure.

In order to obtain the good properties of an FIR filter, however, one gives up the degrees of freedom afforded by the recursive coefficients of an IIR filter. (see prior footnote, the FIR filter is actually a constrained version of the IIR filter) To obtain a similar frequency magnitude response from an FIR filter, large values of N are needed relative to the order of a similar IIR filter. As a result it has been proposed that FIR filters be implemented in the

---

[1] The output of an IIR filter depends on past values of the output as well as past values of the input. The difference equation for an IIR filter is commonly written as follows:

$$y[n] = -\sum_{i=1}^{M-1} a_i y[n-i] + \sum_{i=0}^{N-1} b_i x[n-i]$$

The recursive nature of this computation forces any implementation to be recursive and causes stability to be an issue.

frequency domain using an FFT algorithm to reduce processing requirements. Unfortunately, several of the good properties of an FIR filter are not completely preserved if an FFT implementation is used.

Conventional Implementations

If an FIR filter is going to be implemented in the time domain, the convolution sum has to be computed. The computation requires at least N multiplies and N-1 adds to calculate each output point. In the following discussion I will assume that a sufficient number of multipliers and adders are available to perform the entire computation each time cycle. FIR filters can be implemented to use fewer arithmetic units by time multiplexing[1]; however, these designs directly follow from the complete designs and only could only add unnecessary complication to the discussion.

Because there are a large number of ways to compute a convolution sum, I will discuss the two most common forms in some detail and then give a general introduction to systolic architecture design to characterize the other possibilities.

Direct Form

The textbook FIR filter architecture is the direct form architecture. The direct form design is the result of bluntly translating equation (1) into hardware. A chain of delay registers forms a length N-1 first-in-first-out (FIFO) shift register that stores the previous N-1 (delayed) values of the input. These N-1 delayed values of the input along with the current value of the input are multiplied by the appropriate weights and summed to form the result. The total design contains N multipliers and N-1 two input adders which are the minimum necessary quantities without using a time multiplexing scheme. A length 4 (N=4) direct form filter is shown in figure 1.

---

[1] A time multiplexed design uses the same arithmetic unit(s) more than once per time period with different data. For example, an FIR filter of arbitrary length N could be designed using only one multiplier and one adder; each time period would then be, at minimum, greater than N multiply times. Because we are aiming for the highest throughput possible, it is safe to assume that time multiplexing is not a viable option.

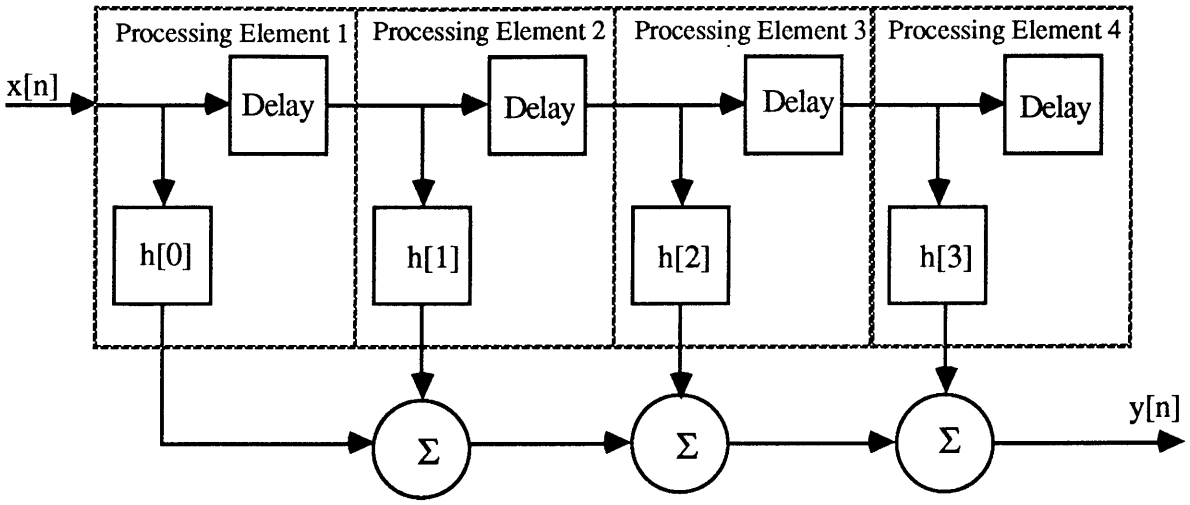


Figure 1 Direct Form FIR Filter

The figure shows N-1 adders in a linear chain for clarity; an actual design would probably use a binary tree of adders (figure 2). The obvious advantage of a tree structure is its reduced latency. The latency of a linear chain is N-1 adder delays; for a tree structure the latency is $\lceil \log_2 N \rceil$ where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. The less apparent advantage is that a tree structure is more easily pipelined. Pipeline registers can be placed on each level of the tree (where the dashed line crosses the data paths in the figure). If a register is placed anywhere between adders in the linear chain, an additional register is needed in all successive adders to delay multiplier results until the correct partial sum arrives.

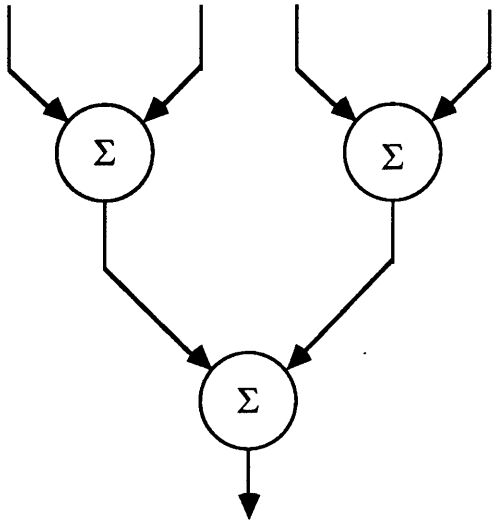


Figure 2 Tree Adder

The tree adder appears to be an obvious choice, however, as with all bonuses there is an equal and opposite penalty attached. The tree adder removes the regular structure that is shown in figure 1. To add an additional stage to the implementation in figure 1, an additional stage is merely attached to the end. To extend a design that uses a tree adder, the entire adder structure must be modified, in this case, a new level must be affixed to the tree. As discussed in the section on systolic architectures, this is not necessarily optimal for VLSI designs; however, it is interesting to note that LSI Logic chose the direct form implementation with pipelining to implement their new FIR filter chip.

Transpose Form

The second most common FIR filter architecture is the transpose form shown in figure 3. At first it is not at all obvious that the two architectures perform the same algorithm, but an easy way to show this is to assume the design correct as drawn and to reverse engineer it. Letting the partial sum out of ith processing element be denoted by $p_i[n-1]$, we can form the following equation for each processing element.

$$p_i[n-1] = p_{i-1}[n-2] + h[n-i] * x[n] \qquad (2)$$

Starting at the first processing element and proceeding inductively, we have

$$p_1[n-1] = h[N-1] * x[n]$$

$$p_2[n-1] = p_1[n-2] + h[N-2] * x[n] = h[N-1] * x[n-1] + h[N-2] * x[n]$$

$$p_i[n-1] = \sum_{j=0}^{i-1} h[N-i+j] * x[n-j]$$

Finally, setting $y[n-1] = p_N[n-1]$ yields the convolution expression in equation (1).

Overall, the transpose form uses more hardware than the basic direct form because the delay registers must be large enough to contain the sums of the scaled versions of the input. Assuming that the coefficients are represented by the same number of bits as is the input, the registers will be twice as wide. If registers are included to the adder tree of a direct form implementation to achieve a similar throughput, the hardware requirements become very similar. The different connection scheme does, however,

significantly change the properties of the design. The largest difference is that all additions are localized so that no "global" N input addition is needed.

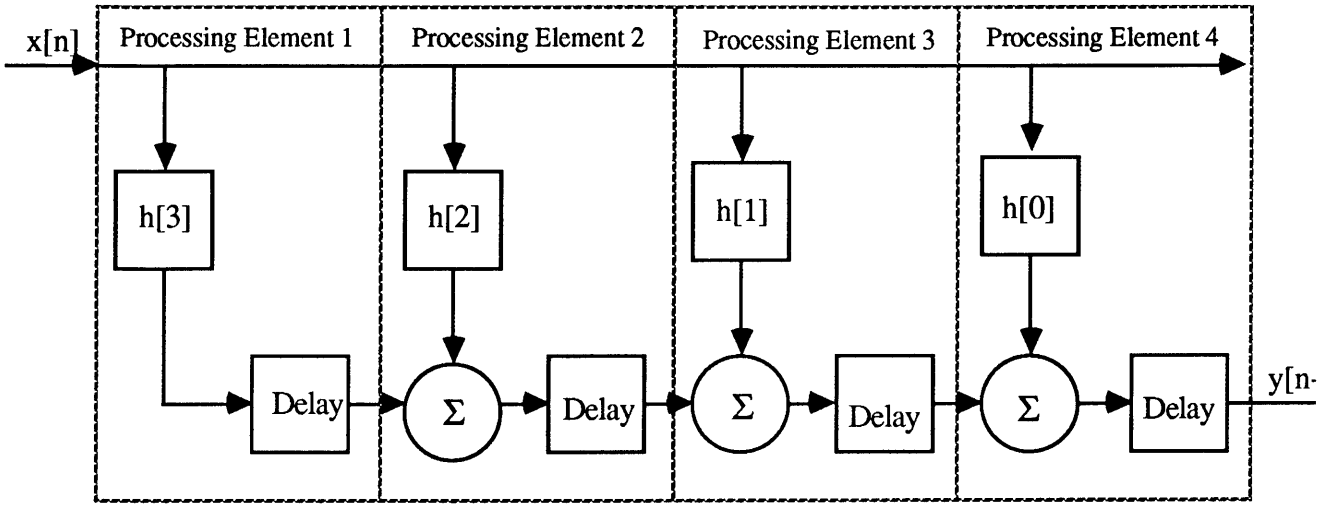


Figure 3 Transpose FIR Filter Form

One result is that the transpose form can be simply expanded by adding additional stages to the end of the linear structure. In addition, the throughput of the transpose form without additional pipeline registers is one multiply delay plus one add delay; the direct form can achieve this throughput also, but only with the addition of pipeline registers to the adder structure. As always there is a cost for these benefits; in this case it is that the input must be broadcast to each tap of the filter. However, as discussed later, this cost may not be too severe.

Systolic Data Flow Graph Architectures

Because FIR filters exhibit a high degree regularity in the computations required, a good framework within which to investigate FIR filter designs is that of systolic architectures. The systolic architecture concepts provide a general methodology for mapping a problem involving a regular computational structure into an architecture. In addition, the concepts provide a general technique for evaluating a possible architecture against other designs.

Systolic architectures became very popular in the late 1970's and early 1980's because of work done at Carnegie-Mellon by H.T. Kung. The basic philosophy of a systolic design is a rhythmic flow of data through a series of

processing elements (PE's). A good analogy for a systolic architecture is a production line where partial results move regularly from one worker to another, each of whom adds something to the final result.[1] The goal is to achieve 100% efficiency (each PE occupied), a simple data flow between PE's, a simple control structure (preferably no control at all), and a modularly expandable design. All of these properties are important for VLSI implementations to minimize design time and maximize system performance. For wafer scale integration[2] where PE's can be selectively interconnected these characteristics become fundamental requirements.

In the early days of systolic architectures there was a drive to push more complex computational problems into the systolic model; however, it is now generally recognized that systolic concepts are better applied to problems exhibiting a very regular computational structure with simple primitive calculations. More complex problems, such as those requiring more general computations, are better mapped to a wavefront architecture[3] or some other more general dataflow architecture. However, there are a number of simple regular problems in signal processing that are ideal to examine within the systolic framework. One of these is the convolution sum.

Because there an infinite number of systolic architectures that could be used to solve a particular problem, a data flow graph can be used to visualize the characteristics of the different possible designs. The idea behind a data flow graph is to list all of the primitive operations of the larger computation to be performed in a geometrically regular grid. As an example, the data flow graph for a four point convolution is shown in figure 4. All of the computation for a single output point is listed on one line; the computation for the next (chronological) output point is listed on the next (consecutive) line.

---

[1] Analogy due to H.T. Kung, one of the fathers of systolic architectures

[2] Wafer Scale Integration (WSI) is a fabrication technique that uses an entire silicon wafer to provide ultrahigh circuit densities. Because the yields for wafer size designs would be unacceptably low, extra processing elements are included, and the top level of metalization is configured to allow selective interconnection of processing elements.

[3] Kung, S.Y. *VLSI Array Processors*, IEEE ASSP Magazine, July 1985, Vol. 2, No. 3, pp 4-22

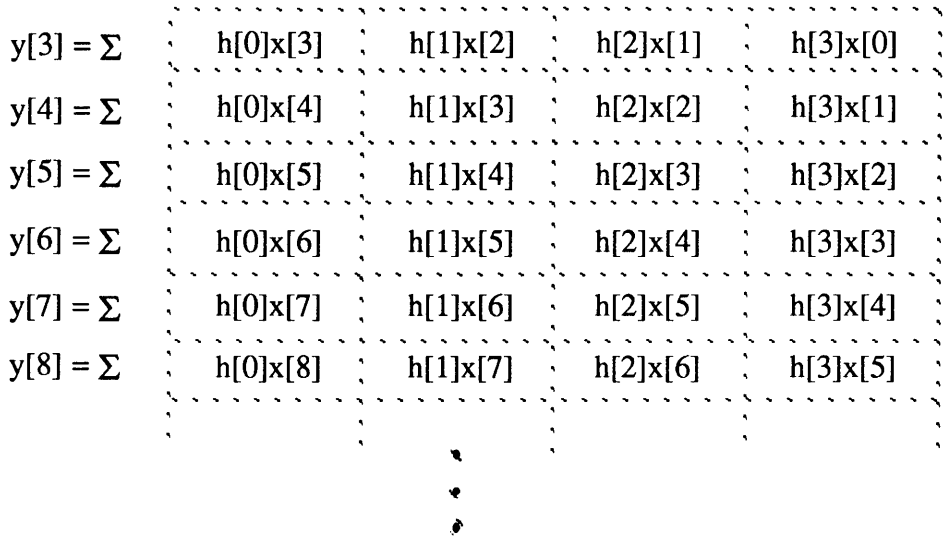


Figure 4 Data Flow Graph

Once the required primitive operations have been laid out in an acceptable manner, the available processors can be assigned primitive operations to perform at each time step. Every pattern of processing which includes the initial processor placement on the data flow graph and the sweep that the processors make across the computations defines a different architecture. Some of these obviously may be more desirable than others. If the processors map to computations on the data flow graph in a linear pattern and are swept in a regular pattern across the data flow graph, the resulting architecture will have a simple data flow between processors and a simple timing for operations.

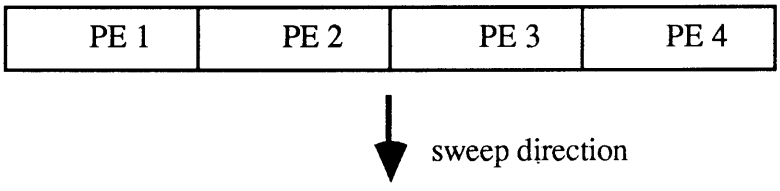


Figure 5 Direct Form Processor Arrangement and Sweep

An example of a processor arrangement and sweep is shown in figure 5. All of the processing for a single output point is performed in a single time period. Each of the PE's, multipliers in this case, computes one of the four multiplies; an additional adder is needed to sum up the results of the four PE's. Moving the row of processors down the data flow graph by one line (one time step) causes the coefficient, h[i], in each processor to remain and causes the

delayed versions of the input to shift right one PE as a new input point enters PE #1. An architecture that has these properties has already been described as the direct form implementation.

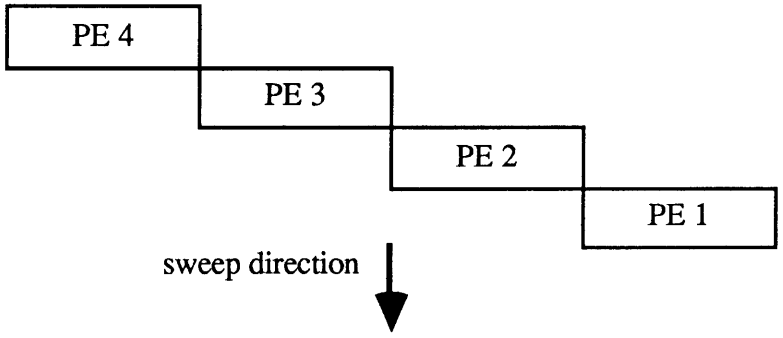


Figure 6 Transpose Form Processor Arrangement and Sweep

Another example of a processor arrangement is shown in figure 6. Because of the diagonal processor arrangement, all of the processors are operating on the same input point at the same time. Also, the coefficients remain in their respective processors from step to step because the sweep direction is downward. Focusing on the computation for a particular output point, PE #1 does the first multiply at time i; PE #2 does the second multiply at time i+1; PE #3 does the third multiply at time i+2; and PE #4 does the final multiply at time i+3.[1] If PE #1 passes the result of the first multiply to PE #2 at time i+1, the two products can be summed to form a partial result that is passed to PE #3 at time i+2. In this way the four PE's, which would be multiply-adders, operate on four different output points at one particular time with results exiting PE #4. This architecture is the same as the the transpose form that was described before and shown in figure 3.

At this point it is possible to characterize the different possible architectures by directly examining the data flow graph rather than trial and error. The placement of the processors determines how the input and/or delayed versions of the input must be made available to computational elements. Two possible processor organizations are seen in the previous examples. A horizontal processor arrangement will require N delayed

[1] At this point the reader should be thankful that a longer FIR filter is not being used as an example

versions of the input to be made available with the current input always used by PE #1 and the oldest version of the input always used by PE #4. A diagonal processor arrangement with slope = -1 will require the current input to be broadcast to all processing elements PE #1 through PE #4. Other arrangements will result in different requirements on the input. For example, a diagonal processor arrangement with slope = 1 would require 2N-1 = 7 delayed versions of the input to be available with every other one used at any time. While the processor layout determines the requirements on the input, the sweep direction determines the requirements on the coefficients. Both the direct form and the transpose form had downward sweeps and therefore in both cases the coefficients remained attached to their respective processing elements. Other sweeps are possible, however. For example, if the processors are swept horizontally, the coefficients circularly shift through the processors each time step.

Although the processor layout and sweep direction specify input requirements and coefficient requirements, respectively, the two must be considered in tandem to determine the data flow requirements between the different processing elements. The easiest way to do this seem to be focusing on a particular output point and examining how the primitive computations for this point are performed. As a final example of a systolic architecture, a design in which the coefficients are shifted through the processor elements will be examined to show the general use of the data flow graph.

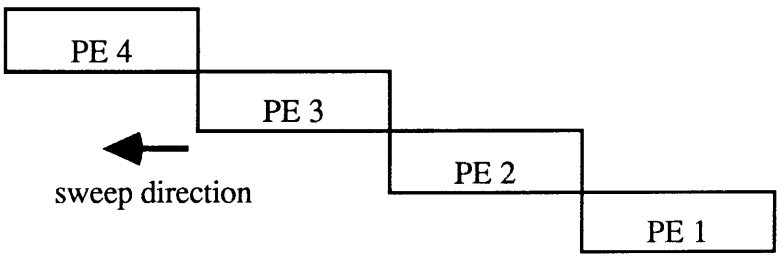


Figure 7 Multiply/Accumulate Processor Arrangement and Sweep

A good exercise is to analyze the processor arrangement and sweep direction shown in figure 7. First, the processor layout, diagonal with slope = -1, indicates that the current input will be broadcast to all processing elements, and the sweep direction dictates that the coefficients will shift through the processing elements. Now, focusing on a single output point, it

becomes apparent that all computation for this output point will be performed by the same processing element over four consecutive time periods. Each processing element, in this case a multiply/accumulator, will accumulate the partial sums and consecutively (as the final product is accumulated) produce an output point. An enhanced version of the multiply/accumulate architecture is used in the Zoran Digital Filter Processor family.

The previous example was chosen because of the interesting advantages to the architecture described. Because the coefficients shift through the filter each time step, an adaptive filter can be implemented that updates the filter coefficients every N time periods. If the output is to be downsampled, the results of all but every Mth processing element can be ignored. In this implementation M-1 out of M processing elements do not even need to be built; all that is needed in these voided places is a register to shift the coefficients by each time cycle. A final advantage is that point failures in an arithmetic unit affect only the output points that would be computed by this element (1 of N output points). But as always there are also disadvantages with this design. First, added control is necessary to determine which processing element should output on a particular clock cycle. In general, this would be solved by adding a tag bit to the coefficient path. A tag attached to h[0] would indicate to a PE to output and clear its accumulator. A second more serious problem would be routing the output from each PE to common output pins for the VLSI chip. Some form of tristate driver arrangement could be used to selectively drive the bus, but the loading on this bus could be excessive.

Although the problems could be solved with the previous architecture, the advantages of the architecture are for a more specialized application; the added complication and hardware is not warranted. For the residue FIR filter I will therefore focus exclusively on the transpose form architecture. This design exhibits a simple modularity, a minimum quantity of hardware required, and a maximum throughput. Other more specific applications that require downsampling or adaptability should probably investigate the multiply accumulate design.

Bitwise Decomposition

As will become obvious later, it is advantageous to minimize the number of residue multipliers that are needed for the residue FIR filter. It is possible to build a binary fixed point filter that uses only adders and no multipliers. A similar design, discussed later, can be used for a residue FIR filter.

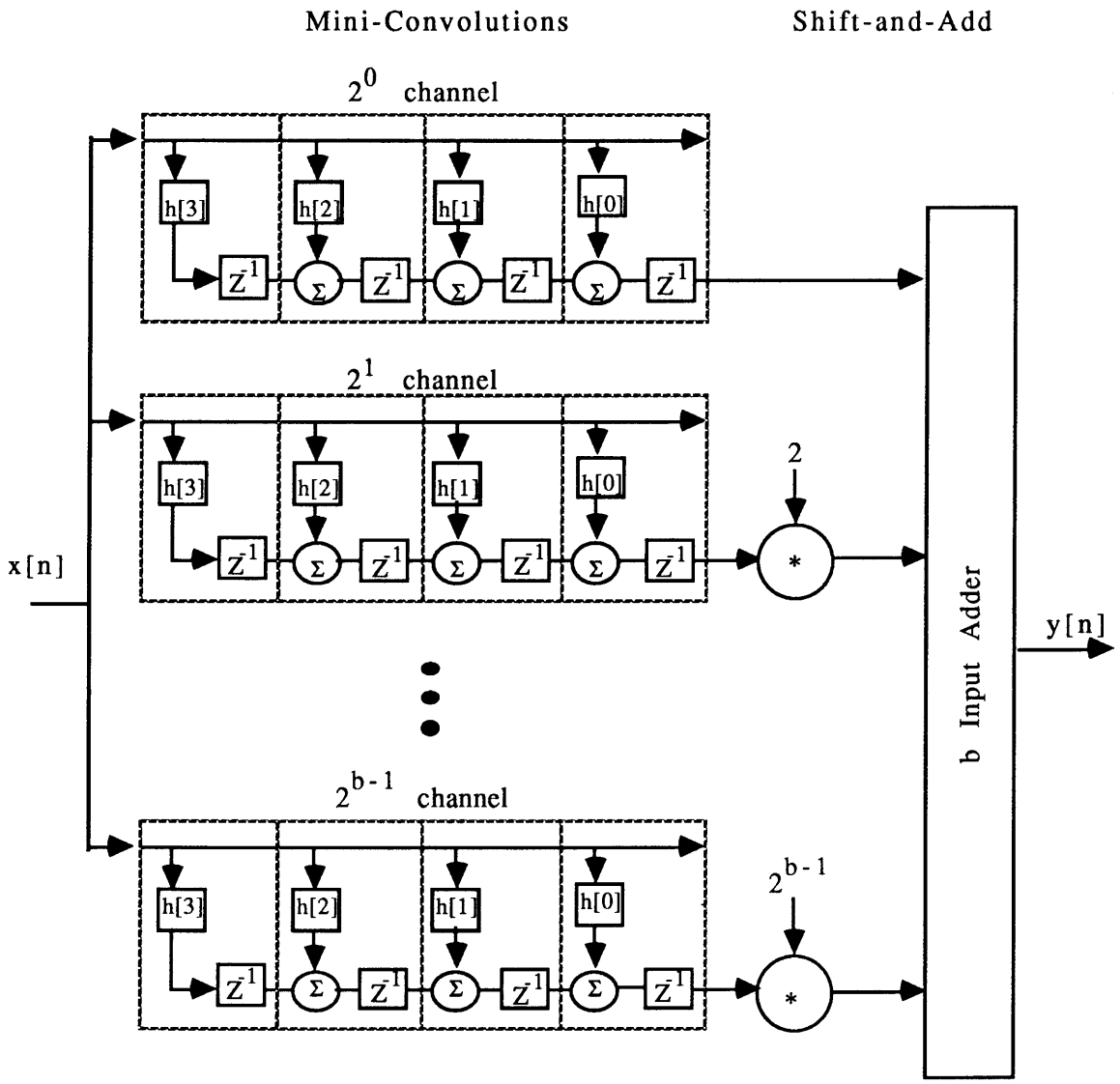


Figure 8 Bitwise FIR Filter Architecture

The idea behind the adder only FIR filter is recognizing that a bxb bit multiply consists of b recursive left shifts and b-1 conditional adds as shown in equation (3).

$$\text{Let } y = \sum_{i=0}^{b-1} 2^i y_i \quad \text{then } x*y = x * \sum_{i=0}^{b-1} 2^i * y_i = \sum_{i=0}^{b-1} 2^i (x * y_i)$$

If the results of several multiplies are being summed then the shifts and adds do not need to be performed until after the final sum. If b is the number of bits needed to represent the coefficients, the convolution equation can be broken into b mini convolution equations the final results of which are shifted and added.

$$y[n] = \sum_{i=0}^{N-1} x[n-i]*h[i] = \sum_{i=0}^{N-1} \sum_{j=0}^{b-1} x[n-1]*h_j[i]*2^j = \sum_{j=0}^{b-1} 2^j \sum_{i=0}^{N-1} x[n-i]*h_j[i]$$

Because $h_j[i]$ in the final equation is either 0 or 1 no general multiplies need to be performed in the mini-convolution chains.

Using the transpose form for the mini-convolutions, the $h_j[i]$'s are employed to condition the adder in each processing element. If $h_j[i]$ equals 1, the current value of the input is added to the previous result; if $h_j[i]$ is 0, the previous result is passed on unchanged. A top level block diagram of bitwise FIR filter is shown in figure 8; an individual processing element is shown in figure 9.

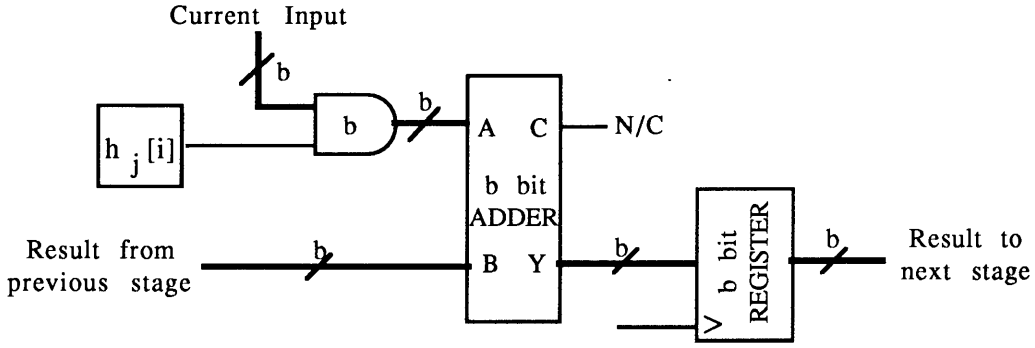


Figure 9 Processing Element for Bitwise FIR Filter

The idea of breaking the FIR computation into parallel channels of accumulators can be extended to other decompositions of the coefficients besides base 2. This concept is used in both RNS designs and the conventional design that follow. A deeper discussion is included in the sections on residue filter design and conventional filter design.

Chapter 3 RNS Background

Residue arithmetic, based on simple principles of number theory, is a possible alternative to conventional arithmetic for larger integer operations. Starting with several relatively prime[1] numbers m_1, m_2, ..., m_r as a moduli set, it is possible to represent an integer x by its residues or remainders to the members of the moduli set: x mod m_1, x mod m_2, ..., x mod m_r. This new representation of x is unique for any integer x that less that the product of the moduli in the moduli set ($0 \le x \le M-1$, where $M = m_1 m_2 \ldots m_r$). The proof of this result is from the Chinese Remainder Theorem.

Residue arithmetic is most useful for the operations of addition, subtraction, and multiplication. A basic result of number theory is that these operations "commute" with the conversion operation: $(((x \bmod m_i) \bullet (y \bmod m_i)) \bmod m_i) = (x \bullet y \bmod m_i)$ where $\bullet$ is either addition, subtraction, or multiplication. The operation is performed independently in each of the r moduli channels; there is no coupling between the channels. Because the moduli can typically be selected to be much smaller than the integers x and y, the operation can be executed in parallel in each moduli channel more rapidly than if x and y are in their conventional representation. It is because of these properties that residue arithmetic is appealing for FIR filters.

Unfortunately, residue arithmetic also has a number of disadvantages. Because division is not a fundamental integer operation[2], it is not straightforward to round or truncate numbers in residue representation. It is equally difficult to compare the magnitude of two numbers. The result of uncoupling the digits in the representation of a number is that there is no longer any significance that can be attached to a particular digit position. In general, a number must be converted back to a conventional representation to perform either of these operations. This leads into the final difficulty: converting into and out of residue representation. The conversion into residue representation is usually done by a table lookup. The conversion out of residue representation uses a result of the Chinese Remainder Theorem and

---

[1] Two numbers are relatively prime if they contain no common integer factors other than 1. For example, the numbers 10 and 21 are relatively prime; 10 and 14 are not.

[2] The set of integers is not closed under division. For example, what integer equals 5 divided by 3?

is not as simple. All of these problems are a topic of current research, but for now let's assume that these difficulties can be overcome and examine possibilities for residue arithmetic units.

Basic Residue Arithmetic Units

In order to design FIR filters, it is necessary to implement some basic RNS arithmetic building blocks. One requirement that will be imposed on these units is that they be "programmable" to permit computations with different moduli. The term programmable implies that the same hardware can be used for any modulus less than a certain size either by rewriting entries into a table or asserting some constant(s) to one or more inputs. If there is a choice between a design that involves a table lookup and one that does not, all else equal, the latter would be preferred. In addition, there are some tricks that can be used with certain classes of moduli to optimize arithmetic computation; however, membership in these classes is fairly restrictive. The overall design will not be practical if specialized arithmetic units must be designed for each modulus. Within these few bounds the goals of the arithmetic units designs are high throughput and minimum size.

Several residue units will be needed in order to implement a RNS FIR filter; although some of these units needed will not be apparent until the actual filter design is begun, it is worthwhile to develop a set of primitive arithmetic units. The techniques used in these designs will be helpful for designing more custom units later. First, the design of a programmable residue adder will be addressed. With the adder the more complex multiply by 2 block and a general multiply block can be designed. Finally, although it involves a table lookup, a general function unit will be discussed briefly.

Residue Adders

One of the earliest proposals for a residue adder was to use a conventional ROM as a table lookup (figure 10). For moduli that can be represented within a b bit binary channel (i.e. $m \leq 2^b$), it is necessary to have a 2^{2b} x b ROM. For example, a 6 bit modulus ($m \leq 64$) would require a 4Kx6 ROM. There was some early research into exploiting the symmetry inherent in the addition tables to reduce to size of the ROM. A first stab is realizing that the operation of addition is commutative; this reduces the size of the ROM by a factor of two. Unfortunately, as the design is optimized to reduce the size of the ROM, the

external circuitry increases and the throughput decreases. In general, the large area required to implement memories and the access time of these memories prohibits this approach for all but the smallest moduli[1]. Although table lookup would not be practical for a residue adder, it is important to note that any integer operation on two variables can be performed using a table lookup.

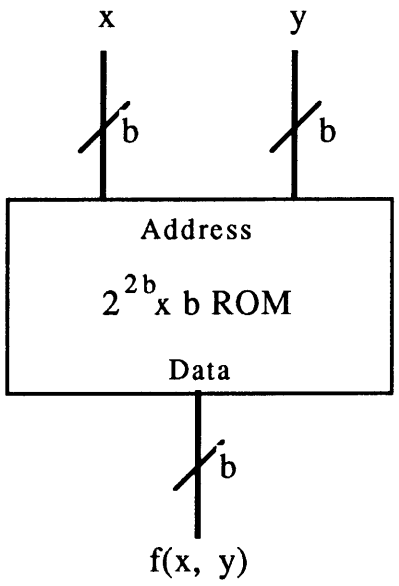


Figure 10 Table Lookup Residue Adder

Focusing on the addition problem, the size of ROM in the previous design can be significantly reduced by using a standard b bit binary adder as shown in **figure 11**. The output of the adder is b+1 bits wide including both the b bit result and a carry bit. Because this b+1 bit output may exceed the modulus, the ROM is necessary to correct the result to lie in the normalized range [0, m-1]. In this case the size of the ROM is 2^{b+1}xb. For a 6 bit modulus, the ROM would be 128x6 bits. Although this is significantly better than the previous design decreasing the size of the ROM by by a factor of 32, a closer examination of the ROM's contents shows that this design can also be improved.

---

[1] Chaing C-L & Jonsson Lennart, Residue Arithmetic and VLSI 1983 IEEE Computer Design? pgs 80-83

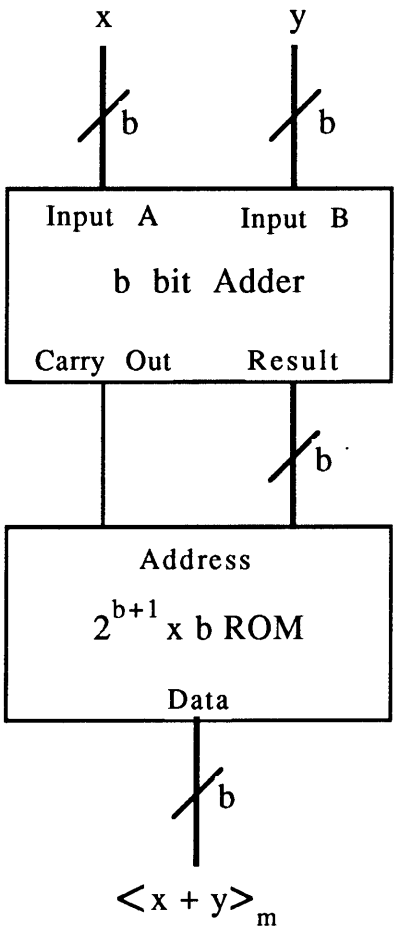


Figure 11 Residue Add using a Binary Adder with Correction ROM

Assuming the inputs to our residue adder are in normalized residue form[1], the output of the binary adder is falls into three cases (see **figure 12**). First, if the sum of the two numbers is greater than or equal to 0 and less than the modulus, the result is already in normalized residue form, and the ROM passes the result unchanged. Second, if the sum of the two numbers is greater than or equal to the modulus and less than or equal to 2^b (where b is the width of the binary adder), the b bit result exceeds its normalized representation by the value of the modulus, and the ROM subtracts the modulus from the output of the binary adder. Finally, if the sum of the two numbers is greater than or equal to 2^b (carry bit set), the b+1 bit result, including the carry bit as the

[1] A residue is in normalized residue form if its magnitude is between 0 and the modulus. A residue is not in normalized residue form if its magnitude is greater than or equal to the modulus or less than 0.

b+1th bit, exceeds the normalized form by the value of the modulus, and the ROM subtracts the modulus from the b+1 bit result.

Modulus $m = 43$ $\qquad 2^b = 64$

Bias $\mu = 64 - 43 = 21$

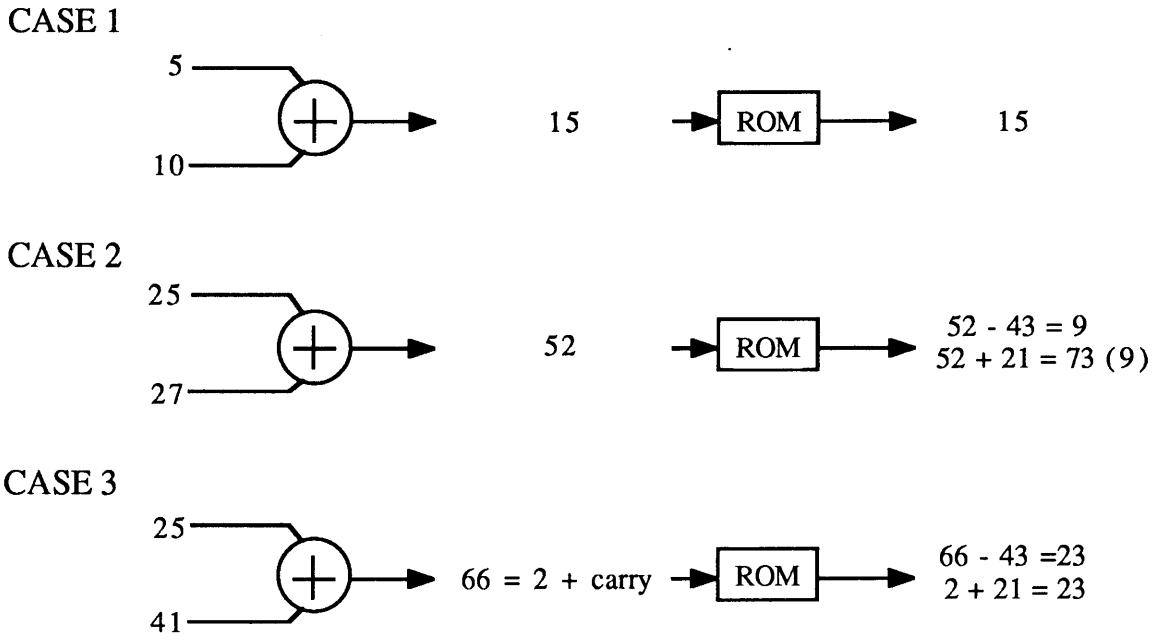


Figure 12 Residue Addition with a Binary Adder

The ROM entries can be reduced to two operations: either the output of the binary adder is passed unchanged or the modulus is subtracted from it. The ROM can be eliminated entirely as shown in figure 13. The first binary adder performs as before with its output that may or may not be normalized. The b+1 bit binary subtracter subtracts the modulus from the result of the first adder. This serves the dual purpose of providing other possible final result and indicating (by its overflow bit) whether the output of the first adder is normalized. If the overflow is set, the output of the binary adder was in the range [0, m-1]; if no overflow is set, the output of the binary adder was greater than m. The overflow can be used to select between the output of the binary adder and subtracter.

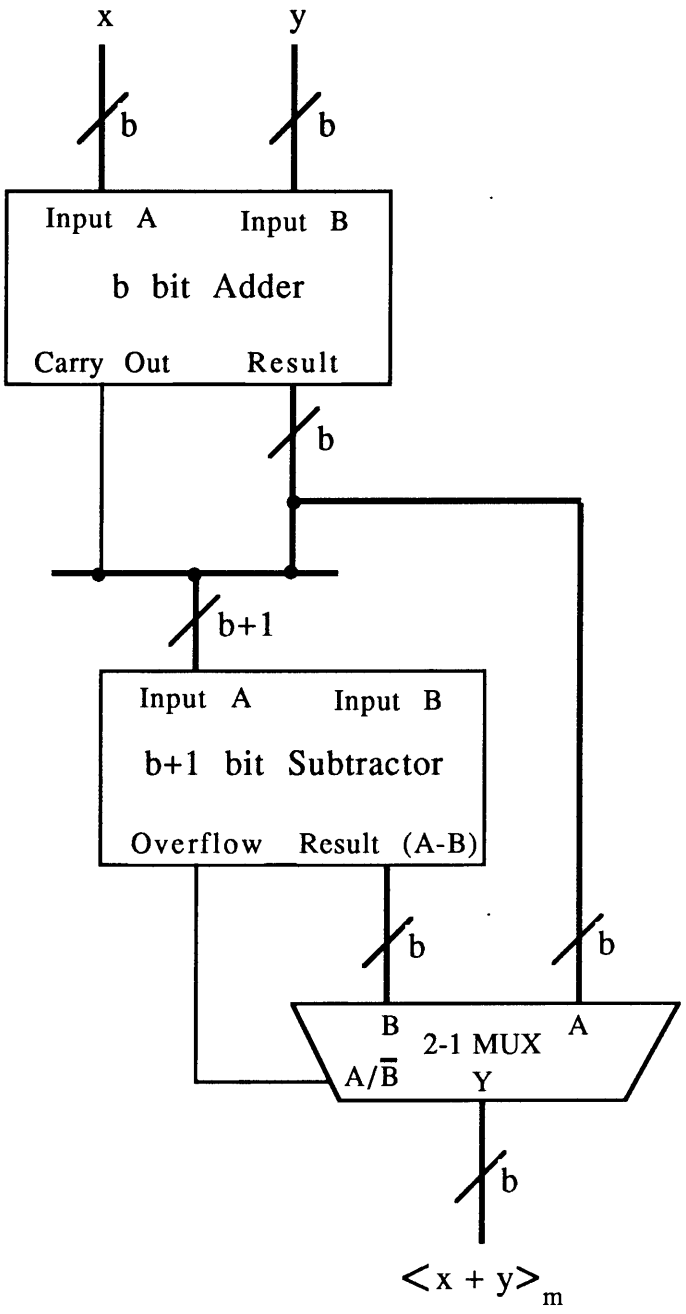


Figure 13 Residue Adder without ROM

One final optimization results from the modulo nature of a finite wordlength binary channel. Adding two normalized residues in the binary adder yields a result, u, in the range [0, 2(m-1)]. If m is representable in a b bit channel, then $2(m-1) \leq 2^{b+1}$. When m is subtracted from u, a number v is obtained that is always less than m (consider only the case $u-m \geq 0$) and is therefore representable in b bits. The number v can be obtained

alternatively by adding $\mu = 2^b - m$ to the low order b bit of u and ignoring the carry; this is a result of the mod 2^b nature of the channel. The advantage to this approach is that a b bit binary adder can be used instead of b+1 bit binary subtracter; one stage of carry propagation is saved. The final residue adder is shown in figure 14.

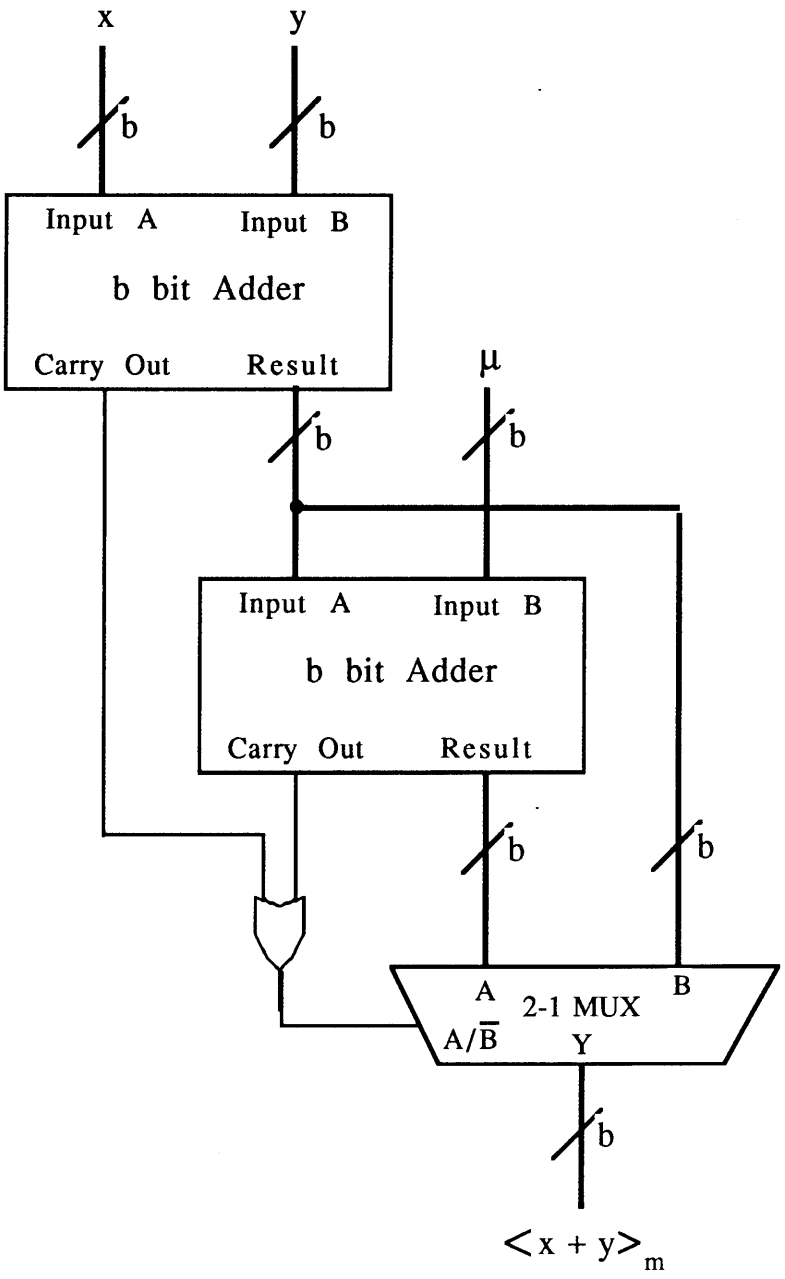


Figure 14 Final Residue Adder Design

Because the carry is not input into the second binary adder, the carry out of the second adder will only be set if u is in the range [m, 2^b-1]. If u is greater than or equal to 2^b (the carry out of the first adder is set), the carry out of the second adder will not be set. The logical OR of the two carries is used to select the multiplexer; if either carry is set, the subtracted version is chosen.

At this point is it useful to include a hardware summary[1] of the final residue adder. Similar summaries will be generated for other final version blocks to permit simple comparisons between more complex architecture sections. The basic components that will go into the summaries are 1 bit full adders, MUX's, and simple gates. For the simple final residue adder the summary is as follows:

Part Type	Number	Sizing	Transistors
1 bit Full Adder	2b	19584b μ^2	64b
2-1 MUX	b	4896b + 1632 μ^2	10b + 4
OR gate	1	4896 μ^2	6
Totals	---	24480b + 6528 μ^2	74b + 10

Architecture Summary for Final RNS Adder

Examining the final modulo adder design, we can gain some helpful insight into the operation of modulo addition performed with binary arithmetic units. The basic result is that modulo addition is the same as binary addition unless the result exceeds the modulus in which case the modulus, m, is subtracted from the binary sum, or, equivalently, $\mu = 2^b - m$ is added to the binary sum. As a result of the previous discussion for the final modulo adder, we will focus on performing the correction, if necessary, by adding μ.

Now, instead of possibly performing the correction later, preadd μ to one of the inputs and use a single binary adder as shown in figure 15. Because x_1 is initially normalized, it falls in the range [0, m-1]; because $x_1 + \mu$ will correspondingly be in the range [μ, 2^b -1], it can be represented entirely in a

[1] The space estimates were derived from an existing standard cell library. The transistor count numbers were derived from simple designs in CMOS and include both p and n type transistors. A more detailed discussion of these hardware estimates is included in the Appendix.

b bit binary channel, and the carry out of the preadder can be ignored. By the previous result the output of the main binary adder will now either equal the correct modulo sum or exceed this sum by μ. The carry out of the binary adder provides a flag to indicate which case the answer is in. If $x_1 + x_2$ is greater than or equal to m, then $x_1 + x_2 + \mu$ will be greater than or equal to 2^b and the carry will be set. Since $x_1 + x_2 \geq m$ is the case that needed correction, the output of the binary adder, ignoring the carry, is the proper normalized modulo sum. If $x_1 + x_2$ is less than m, then $x_1 + x_2 + \mu$ will be less than 2^b and the carry will not be set; the output of the binary adder will exceed the correct normalized modulo sum by μ.

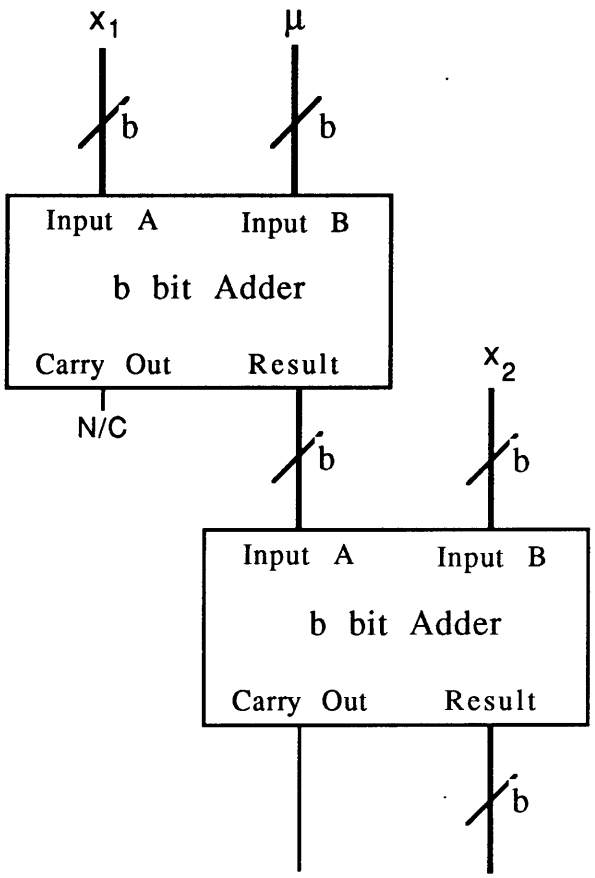


Figure 15 Preadding μ to one of the Inputs

At first it appears that preadding one of the inputs trades one problem for another very similar problem. Without preadding, the binary sum, ignoring the carry, can fall short of the correct modulo sum by μ; with preadding, the binary sum can exceed the correct modulo sum by μ. However, if a series of

numbers x_i is being accumulated and the preadd of μ to each can be performed at minimal expense, we can increase the performance over that of the general residue adder. It is always possible to guarantee that one of the inputs to the binary adder is a biased residue (exceeds its normalized value by μ) and the other is a proper normalized residue. If the current partial sum is a biased residue, the carry out, 0, is used to select the normalized version of x_i; if the current partial sum is normalized, carry out, 1, is used to select the biased version of the input. The completed modulo accumulator including a necessary register is shown in figure 16.

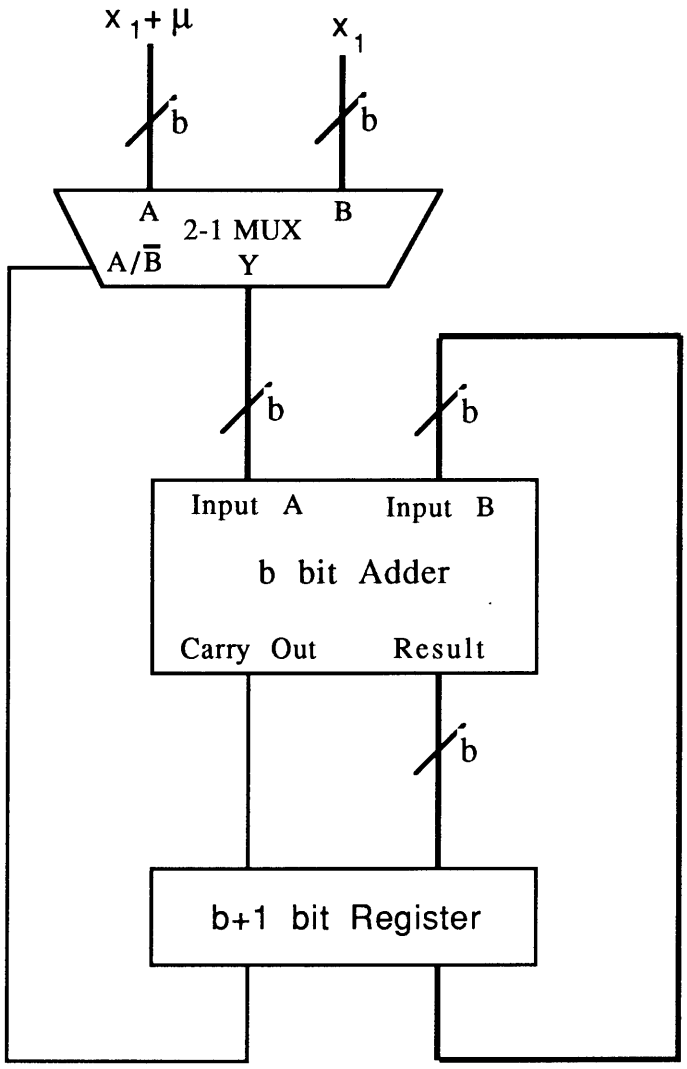


Figure 16 Residue Accumulator

A single addition in the modulo accumulator is requires only one b bit adder delay, while an addition in the final modulo adder requires two b bit adder delays. On the surface it appears that the modulo adder could be improved somehow; however, it is important to realize that the accumulator is a rather constrained form of the addition problem. Also, the one b bit adder delay assumes that the preadds can be performed with no overhead and is an only average delay value. An additional correction stage must be included at the output of the accumulator because the final sum may be in biased form. Nevertheless, even with all of these caveats, this configuration is very useful when several numbers are being accumulated (for example an FIR filter).

Residue Multiply by 2 Block

The next arithmetic unit to examine is a modulo multiply by 2 block that takes a normalized residue input and generates a normalized residue output. This block is very useful when building the more complex general modulo multiplier block. Now, in the standard binary number system it is simple to multiply a number by two: simply shift left one place. Unfortunately, nothing is as straightforward in residue computations and this is no exception.

The obvious way to implement a modulo multiply by two block is to build upon what we already know by using a modulo adder with both inputs tied together. Looking at the final modulo adder in figure 14, the output of the first b bit adder will just be a left shifted version of the input. The left shift function, however, can be hardwired making the first adder unnecessary. To eliminate the adder, the high order bit of the input is routed to "carry out," and the remaining b-1 bits are left shifted with a 0 inserted as the low order bit to form the b bit "result." The basic multiply by two block is shown in figure 17.

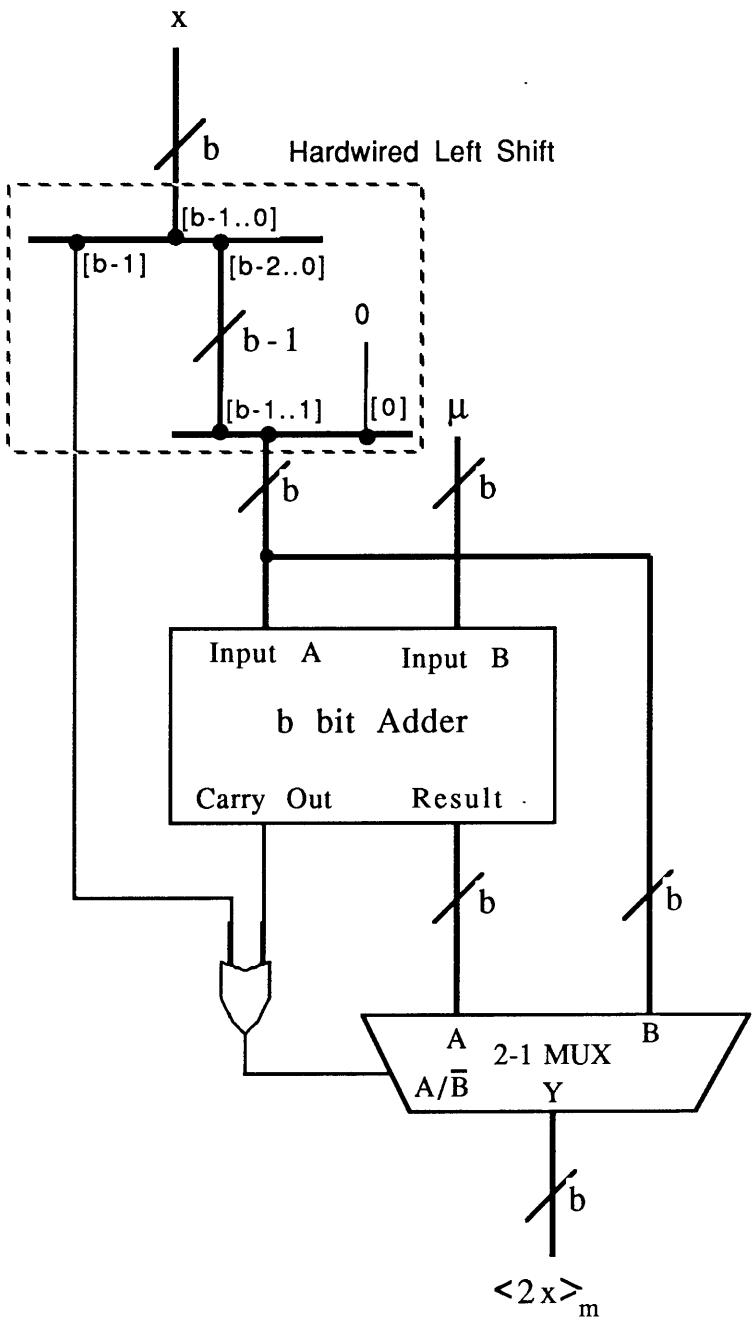


Figure 17 Residue Multiply by 2 Block

Residue Multipliers

The final residue arithmetic block to be added to our toolbox is a general modulo multiplier. A multiplier block is considerably more complex than the adder block or multiply by two block. Because several modulo multipliers are needed in the RNS to binary converter, the general overall system design

goals for latency, throughput, and hardware real estate must be addressed. Hopefully, the multipliers can be designed in such a way to prevent them from being the system bottleneck.

At this point it is instructive to investigate the design of standard binary multipliers[1] before tackling the more complicated modulo multiplier problem. Binary multiplier designs can be divided into two classes: shift and add multipliers and array multipliers. Shift and add designs are by their nature clocked and tend to be slower overall; array designs which are not necessarily clocked (although pipeline registers could be inserted into carry chains) are faster because carries are propagated more efficiently.

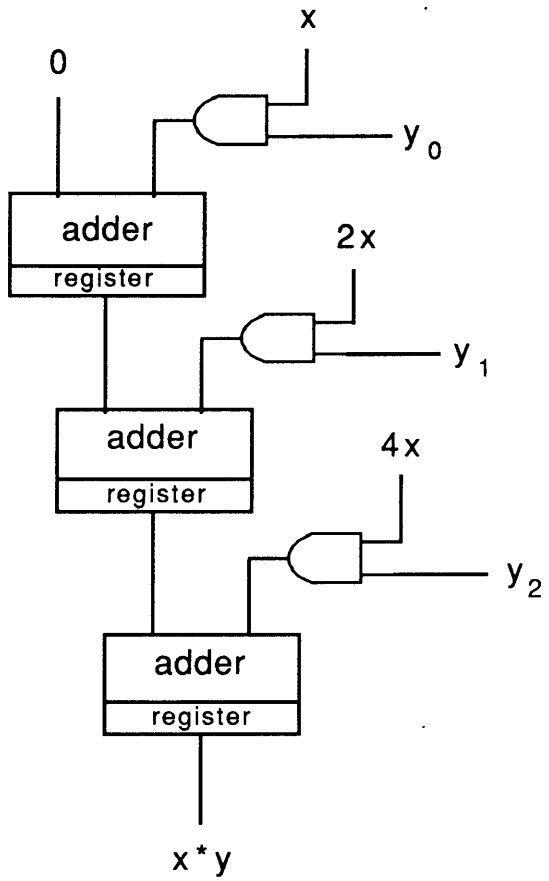


Figure 18 Shift and Add Multiplication

[1] Material in this section was obtained from Rabiner and Gold *Theory and Application of Digital Signal Processing*, pgs 514-540. See this reference for a more exhaustive discussion of binary multiplier design. Other ideas can be obtained by using the systolic design techniques discussed earlier.

A shift and add multiplier forms its product exactly as its name implies by accumulating the following sum:

$$\text{Let } y = \sum_{i=0}^{b-1} 2^i y_i \quad \text{then } x*y = x * \sum_{i=0}^{b-1} 2^i y_i = \sum_{i=0}^{b-1} (2^i * x) * y_i$$

Shifted values of the multiplicand x are accumulated conditioned on the appropriate bits of the multiplier y. An unwrapped nonrecursive version of the shift and add multiplier is shown in figure 18.[1]

The major disadvantage of a shift and add multiplier is that the carry bits do not propagate efficiently. To solve this problem array adders attempt to minimize the length of the longest carry propagation path. A simple 3x3 bit array multiplier is shown in figure 19. In the figure the circles represent 1 bit full adder cells. Better array multipliers can be created using more complicated carry propagation schemes, but the basic structure remains the same with n^2 full adders and the simpler version is easier to understand.[2]

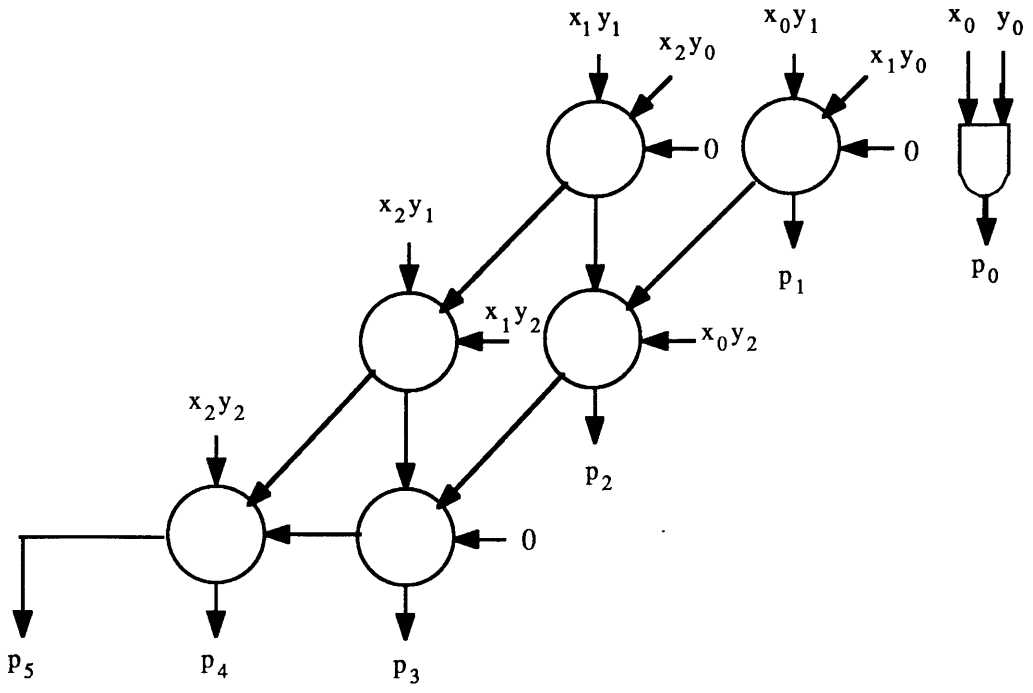


Figure 19 3x3 Array Multiplier

---

[1] A design that uses a smaller adder and is recursive is shown in Rabiner and Gold pg 516

[2] Again, see Rabiner and Gold for a discussion of various implementations.

With some knowledge of binary multipliers, we are ready to attack the modulo multiplier problem. Of the two classes of binary multipliers, the shift and add type seems most conducive to the modulo problem because the partial accumulations can be normalized after each step. Although the array multiplier would be faster, the result of a full bxb bit binary multiply could exceed the modulus by several times its value. In order to normalize this result, several stages of correction circuitry would be needed.

In order to implement a modulo shift and add multiplier, both residue multiply by two blocks and residue adder blocks must be available. Although versions of both blocks have been designed, the final residue adder unfortunately has almost twice the latency of the multiply by 2 block. If a multiply by two block could be designed that provided both the normalized and biased versions of the output, a structure similar to the accumulator in figure 7 could be used that would have the reduced latency that we desire.

An enhanced version (figure 20) of the multiply by two block can be designed with minimal hardware cost and no additional delay. To understand the biased side of the multiply by two block, two cases must be examined, $2x[n] \geq m$ and $2x[n] < m$. Regardless, the binary output of the left shifter equals $2x[n] + 2\mu$. If $2x[n] \geq m$, then $2x[n]$ is an unnormalized residue and one of the two μ's is needed to normalize the residue yielding the desired result, $2x[n] + \mu$. If $2x[n] < m$, then the output of the left shifter exceeds the desired result by μ. Fortunately in the case $2x[n] < m$ the output of the adder in the unbiased side of the doubler has the desired result. The additional components required for this modification are a b bit 2-1 MUX and a b bit register. The only disadvantage of the design is that both biased and unbiased versions of the input must be available, but if several multiply by two blocks are chained together, this is only a problem for the first one.

The modified accumulator will be very similar to the previous accumulator except that the accumulation will not occur in place (not recursive) and the add is conditional. The accumulator will take partial results from the previous stage, add another value to the partial sum and pass the result to the next stage. The carry out from the previous stage must also be passed to indicate whether the partial result is biased. The appropriate bit of the multiplier, y_i, must also be asserted to determine whether to perform the add or pass the previous partial result to the next stage.

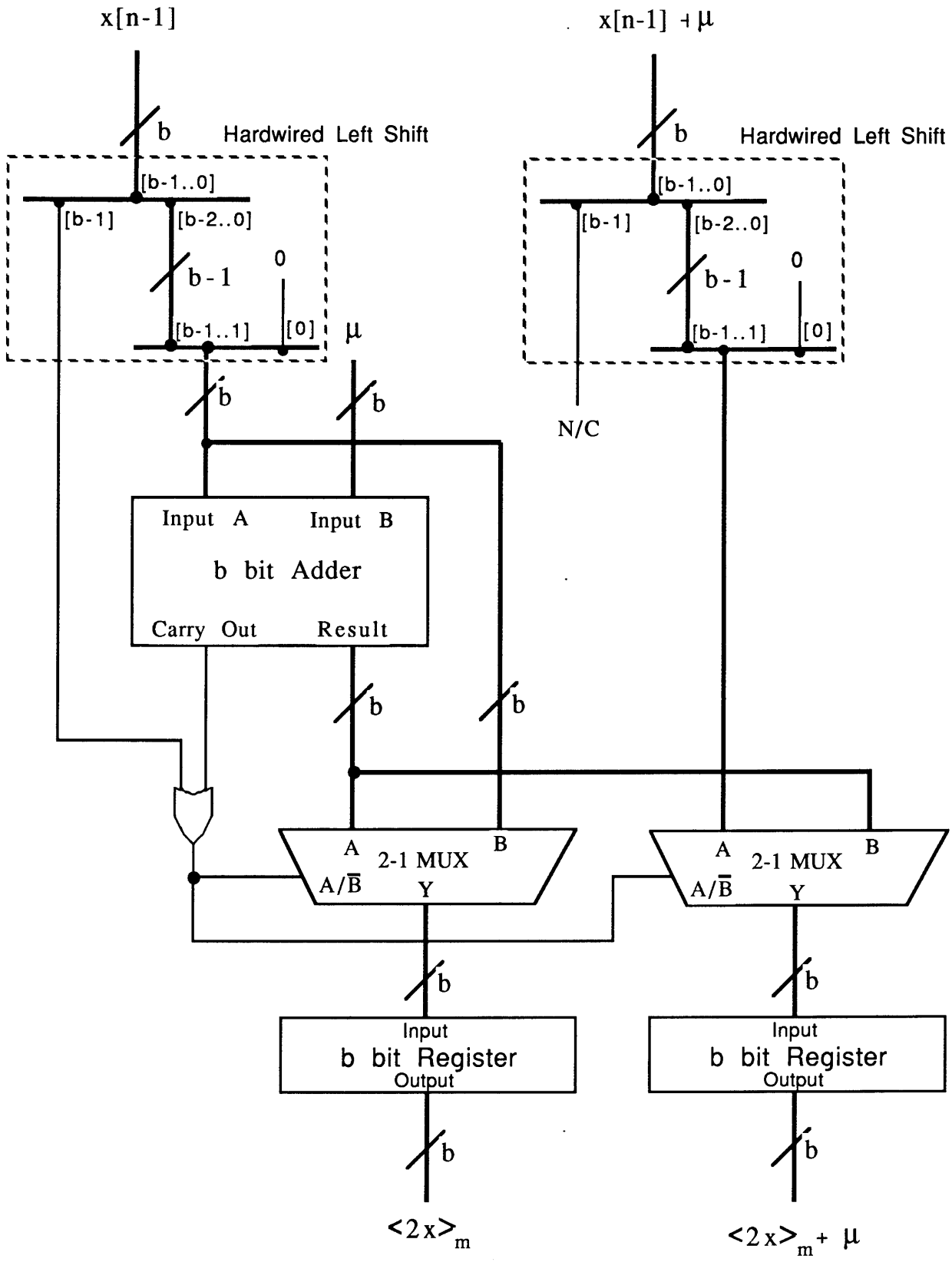


Figure 20 Modified Multiply by 2 Block

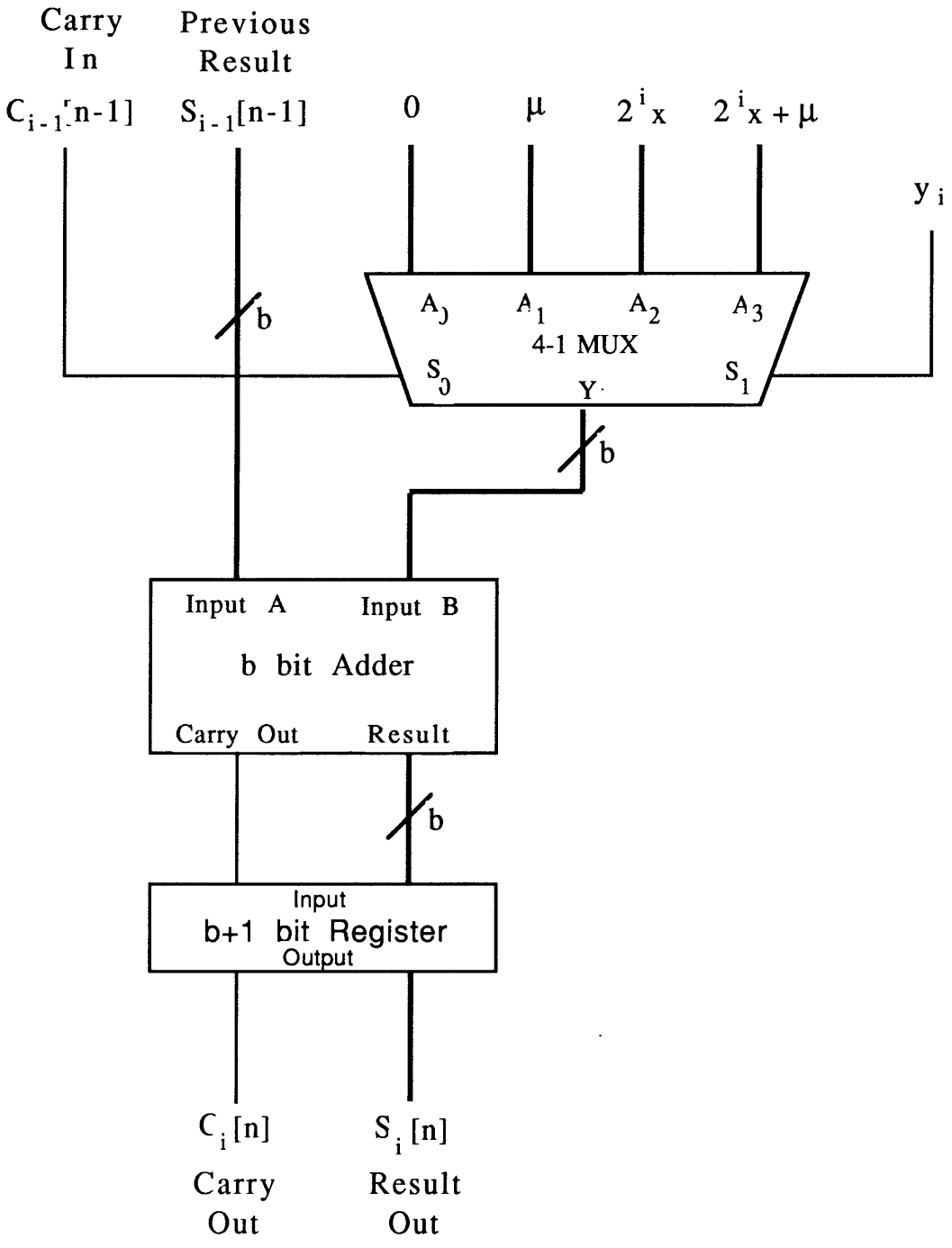


Figure 21 Residue Conditional Accumulator #1

A possible design for this accumulator is shown in figure 21. The 4-1 MUX in the figure performs the following function:

y_i	Cin	MUX output	Explanation
0	0	0	previous result biased, result same
0	1	m	previous result unbiased, result biased
1	0	Input	previous result biased, use unbiased input
1	1	Input + μ	previous result unbiased, use biased input

Both biased and unbiased versions of zero are needed to assure that the carry out of the accumulator properly indicates the state of the result. If the previous result is unbiased and a 0 is added to it, there will be no carry from the adder and the next stage will assume a biased input. Instead, a biased zero is added to the unbiased previous result which biases the result; the carry out again will not be set, however this time, correctly indicating that the result is biased.

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b	9792b μ^2	32b
4-1 MUX	b	8160b + 9792 μ^2	18b + 32
1 bit Register	b+1	8160b + 8160 μ^2	24b + 24
Totals	- - -	26112b + 17952	74b + 56

Hardware Summary for Residue Accumulator #1

Unfortunately, it is not aesthetically pleasing to use rungs of a multiplexor to decode zeros or to require a biased zero to be added to the previous result. If the carry out from the accumulator can be fixed up, the multiplexor can be reduced to a 2-1 multiplexor by placing an AND gate after the multiplexor to condition the add. The biased zero was only necessary in the case when the no add is being performed ($y_i = 0$) and the previous result is unbiased ($C_{in} = 0$). If the previous result is passed to the next stage unchanged, the carry out must be set to 1. With the addition of the necessary gates, the new accumulator design is shown in figure 22.

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b	9792b μ^2	32b
2-1 MUX	b	4896b + 1632 μ^2	10b + 4
1 bit Register	b+1	8160b + 8160 μ^2	24b + 24
AND gate	2	9792 μ^2	12
OR gate	1	4896 μ^2	6
Inverter	1	3264 μ^2	2
Totals	---	22848b + 27744	66b + 48

Hardware Summary for Residue Accumulator #2

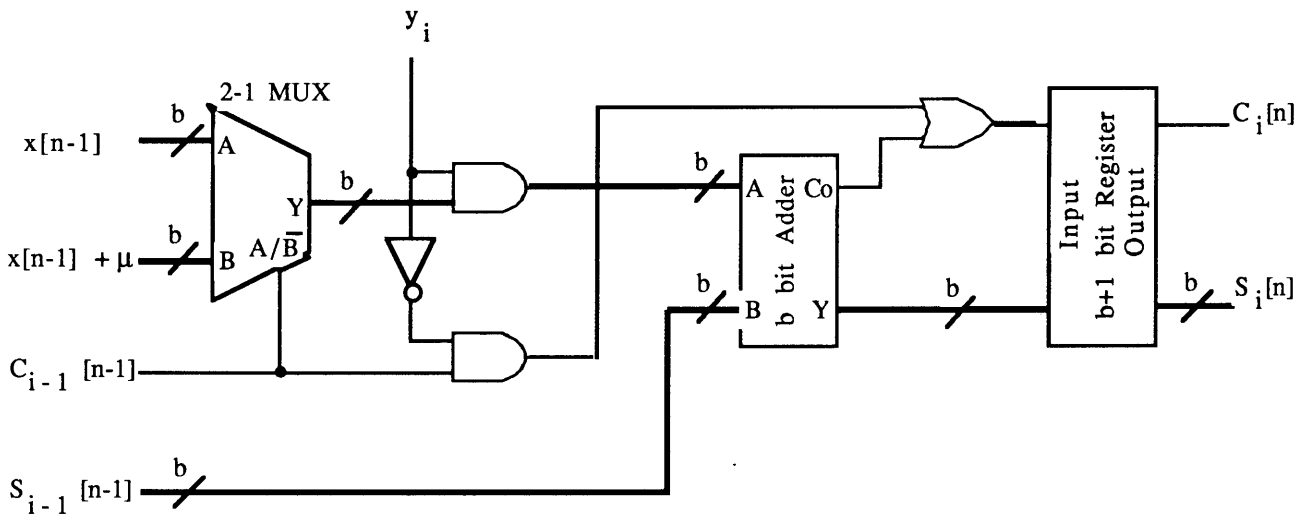


Figure 22 Residue Conditional Accumulator #2

Finally, we are ready to put all of the blocks together to form the general purpose programmable residue multiplier. The only block in the multiplier which has not been exhaustively discussed is the "first stage" which contains a b bit adder that generates a biased form of x[n] and two b bit pipeline registers. Because there is no previous result the first section of the multiplier does not need an accumulator. b AND gates determine whether an unbiased 0 or an unbiased x[n] is passed to the second section. The carry in of the second section accumulator is accordingly hardwired to 1 to expect an unbiased previous result. A final section that unbiases the potentially biased output of the final accumulator has not been included, but may be needed.

The complete multiplier requires a large amount of circuitry. In addition to the first section there are b-1 other sections, the ith one of which requires a b bit doubler, a b bit accumulator, and a b-i bit register. Using either accumulator design, the number of 1 bit full adders needed for a bxb bit residue multiplier is $2b^2 - b$. This can be compared to the b^2 1 bit full adders needed for the binary array multipliers discussed earlier. An abbreviated summary of the residue multiplier is shown below.

Part Type	Number
1 bit Full Adder	$2b^2 - b$
2-1 MUX	$3b^2 - 3b$
1 bit Register	$3.5b^2 + 1.5b - 1$
AND gate	$b^2 + b - 1$
OR gate	$2b - 2$
Inverter	$b - 1$
Totals	---

Hardware Summary for complete Residue Multiplier

On the positive side the residue multiplier has fairly high performance. The latency through the multiplier is b+1 clock cycles, and the throughput is 1 clock cycle. The limiting factor on the clock cycle time is the delay through the modulo accumulator consisting of a b bit binary adder delay and a few gate delays. Even with the performance that can be obtained, the amount of hardware required encourages any residue design to avoid multipliers if at all possible.

Residue General Function Units

For more general functions it is worthwhile to examine table lookup approaches. To implement a general integer function, it may be more efficient, in some cases, to use table lookup rather than a more complex combination of binary arithmetic units. The general function of two b bit variables has already been discussed as the first possible adder design. Here a 2^{2b} x b ROM was used which is the most general implementation of an integer function of two variables. If the function exhibits any special properties, the

table lookup can be broken into several smaller pieces. In this case the throughput and hardware requirements of the table lookup approach may become competitive with those of custom designs.

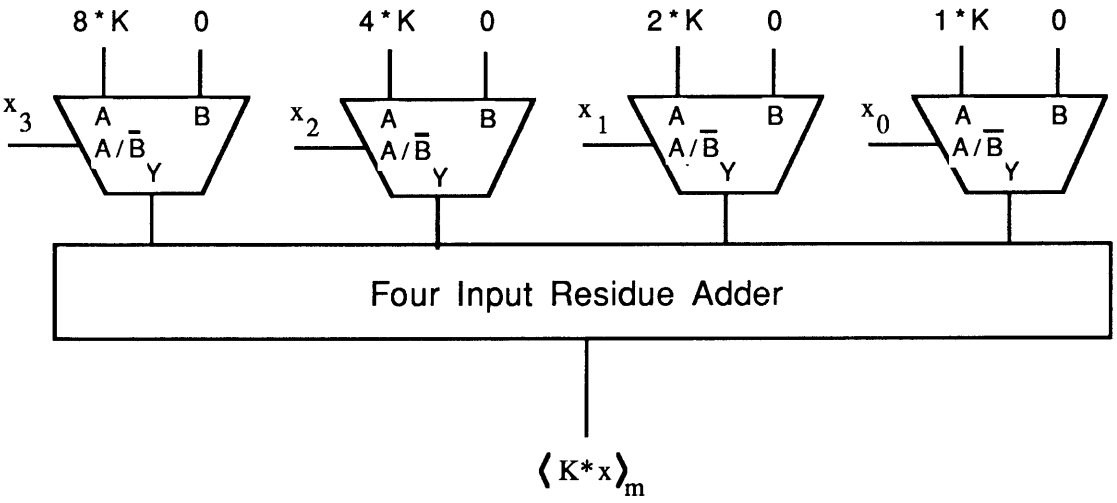


Figure 23 Four bit General Linear Function Evaluation (d=1)

First, let's examine the case of a function of one variable. If the function is linear[1], the b bit input can be broken into groups of d bits, and $\lceil b/d \rceil$ smaller d bit lookups can be performed with the results added. For example to scale a four bit residue, x, by a four bit constant, K, the bits of x can be used to select precalculated multiples of the constant k from tables or hardwired to multiplexors. This is shown more clearly in figure 23 for the case d = 1. Instead of selecting between 0 and a multiple of K, the 2-1 MUX's could be replaced with AND gates when d=1. If d>1, MUX's or some other table addressing scheme will have to be used. An example of d=2 is shown in figure 24. It is interesting to notice that with d=1 the design is very similar to the general residue multiplier except that since K is known, multiples of it can be precalculated.

The evaluation of a multivariable function could also use this trick. However, for an integer function of two variables to be performed by partial

[1] A linear function is one which satisfies the equation, f(ax + by) = af(x) + bf(y). An example of a linear function is f(x) = Kx; an example a function which is not is f(x) = x + K.

lookups, the function must also be linear.[1] Considering that a linear function of two variables has the property f(x,y) = f(x + y) = f(x) + f(y), the two inputs could be evaluated in parallel and the results added to obtain the final result. So, in general, the partial evaluation scheme is most useful for single variable functions.

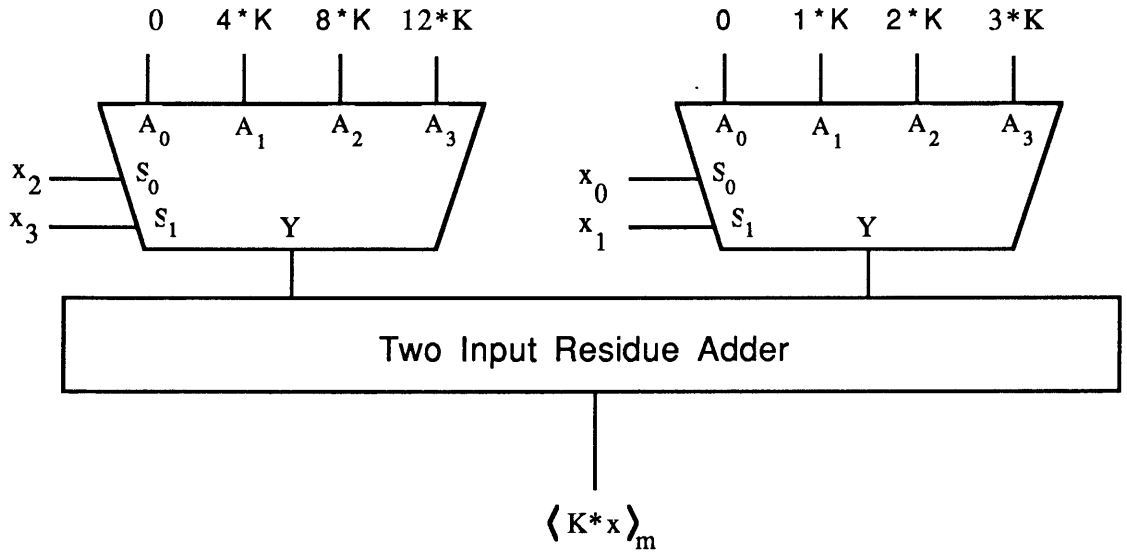


Figure 24 Four bit General Linear Function Evaluation (d=2)

Problems with RNS

The discussion to this point has focused on the advantages of RNS, how parallelism is added to the computation and how modulus programmable arithmetic units can be developed using standard binary arithmetic units. Unfortunately, there are also several disadvantages of RNS. First, binary numbers must be converted into their residue form, and results must be converted back to binary. The substantial size and latency of the conversion units tend to rule out the use of RNS for all but very large computational tasks. Second, because the residue digits are uncoupled, a number in residue form cannot be scaled in any simple manner; all digits must be modified consistently. Finally, also because the residue digits are uncoupled,

[1] Actually, the function has to satisfy a slightly weaker requirement than linearity in two variables. This requirement is as follows: f(x,y) = f(X,Y) + f(X-x, Y-y) $\forall$ (X,Y) within the range of f. The author challenges the reader to find a useful function satisfying the above condition that is not linear.

magnitude comparison is not possible in the residue form. These problems with RNS drastically limit the number of possible applications and to some extent explain the reason that RNS has not been widely used.

Conversion into and out of residue representation

The conversion into residue representation is simple and the conversion for each modulus can operate independently. The residue mod m of a number is the remainder obtained when the number is divided by the modulus. If r moduli are included in an RNS design, r similar conversion units can operate in parallel.

The conversion out of residue is not simple and the conversion process cannot operate independently for the residue from each modulus. The conversion is based on a classic theorem known as the Chinese Remainder Theorem (CRT)[1]. Given the residue representation $\{x_1, x_2, ..., x_r\}$ of x, the value of x can be computed using the following identity:

$$x = \left(M_1 \langle M_1^{-1} \rangle_{m_1} x_1 + M_2 \langle M_2^{-1} \rangle_{m_2} x_2 + \ldots + M_r \langle M_r^{-1} \rangle_{m_r} x_r \right) \bmod M$$

where $M = \prod m_i$, $M_i = M/m_i$, and $x_i = x \bmod m_i$. Although conversion hardware will be addressed in section 4.3, the large mod M computations do not look very promising.

To avoid mod M calculations, the Mixed Radix Conversion (MRC) algorithm is conventionally used. Although the details of the algorithm will be discussed in a later section, it is a two part conversion process that passes through an intermediate mixed radix representation.[2]

Scaling and Magnitude Comparison

Because RNS is not a weighted number system, it is not possible to round a number or to compare the magnitude of one number to another. For either of these operations the residue digits must be converted out of the residue

---

[1] In fact, the mathematical validity of the residue number system relies on the results of the theorem.

[2] Mixed radix number systems are similar to fixed radix except that the radix α can vary from place to place. If x_i are the digits of a mixed radix number with radices α_i, the value of the number is computed as follows:

$$x = \sum_{i=0}^{J-1} \left(\prod_{j=0}^{i} \alpha_i \right) x_i$$

representation. Rather than converting to a fixed radix number, the first part of the mixed radix algorithm can be used to convert to a mixed radix representation which itself is a weighted number system. All of the operations for the conversion can be performed within the modulo channels, and the algorithm can be reversed to return to a residue representation.

Processing Complex Quantities (QRNS)

To this point only real integer computation has been considered; however, in many high signal filtering applications both complex data and coefficients are encountered. Several approaches have been developed to deal with complex residue representations, but these approaches tend to fall into two general categories. The first is simply to use three parallel real residue channels. The second is to use one of the quadratic residue number systems (QRNS).

Processing complex quantities with three parallel channels is performed using an innovative trick. If (a + bi) is the complex input and (c + di) is a coefficient, the inputs to the three real channels are a, b, and a+b, and the coefficients of the three channels are c, d, and c+d, respectively. The output of the second channel (bd) is subtracted from the output of the first channel (ac) to form the real part of the result (ac - bd). The outputs of both the first (ac) and the second (bd) channel are subtracted from the output of the third channel (ac + bc + ad + bd) to form the complex part of the result (ad + bc).

The hardware expense of adding a third channel can be significant, however. To avoid this expense the quadratic residue number system (QRNS) has been developed that can uncouple the real and imaginary part of complex operations.[1] QRNS is a complex modulus number system isomorphic to the extension fields of primes of the form 4k + 1 where k is an integer. For primes of this form, -1 is a quadratic residue.[2] Letting I represent the quadratic residue of -1 modulo p, the two quadratic residues of the input (a+bi) are formed as follows: A = (a + bI) mod p and B = (a - bI) mod p. With the coefficients in a quadratic residue form also C = (c + dI) mod p and D = (c + dI)

---

[1] For more detail on QRNS see the reference section for some interesting papers.

[2] A number r is a quadratic residue modulo p iff there is a solution to the equation

$$x^2 \equiv r \mod p$$

mod p, multiplication and addition are uncoupled: (A, B) • (C, D) = (A • C, B • D) with the computations A•C and B•D being performed in modulo p channels.

At first QRNS seems to be an advantage over using three conventional residue channels, but deeper investigation reveals that the advantage is not as significant as expected. Although only two channels are used to represent complex numbers, the moduli in each channel come from a significantly limited set. The moduli in a QRNS system must be primes of the form 4k+1. The moduli in a standard RNS system only need to be relatively prime to one another. This limitation causes a much smaller dynamic range from a set moduli. To avoid this problem other number systems have been developed such as modified quadratic number system (MQRNS) that allow a richer set of moduli, but these systems have other problems. For a more detailed discussion of QRNS and its extensions see the references listed at the end of the thesis.

Chapter 4 Modular Efficient RNS FIR filter

The architecture discussion that follows focuses on a real integer FIR filter. Although implementations can be derived that include complex coefficients and data using the techniques from the previous chapter, the resulting hardware designs are very similar. Computing with complex quantities either increases the number of residue filter channels and/or slightly complicates the conversion into and out of residue representation. The inclusion of complex cases would only add unnecessary confusion.

It is also assumed that the designs will be implemented in VLSI or WSI. Practically, there is no other platform that could support the massive hardware required for the RNS designs. However, this assumption places some restrictions of the interconnectivity between the different blocks in the design. Some of these have been mentioned in the section on systolic designs.

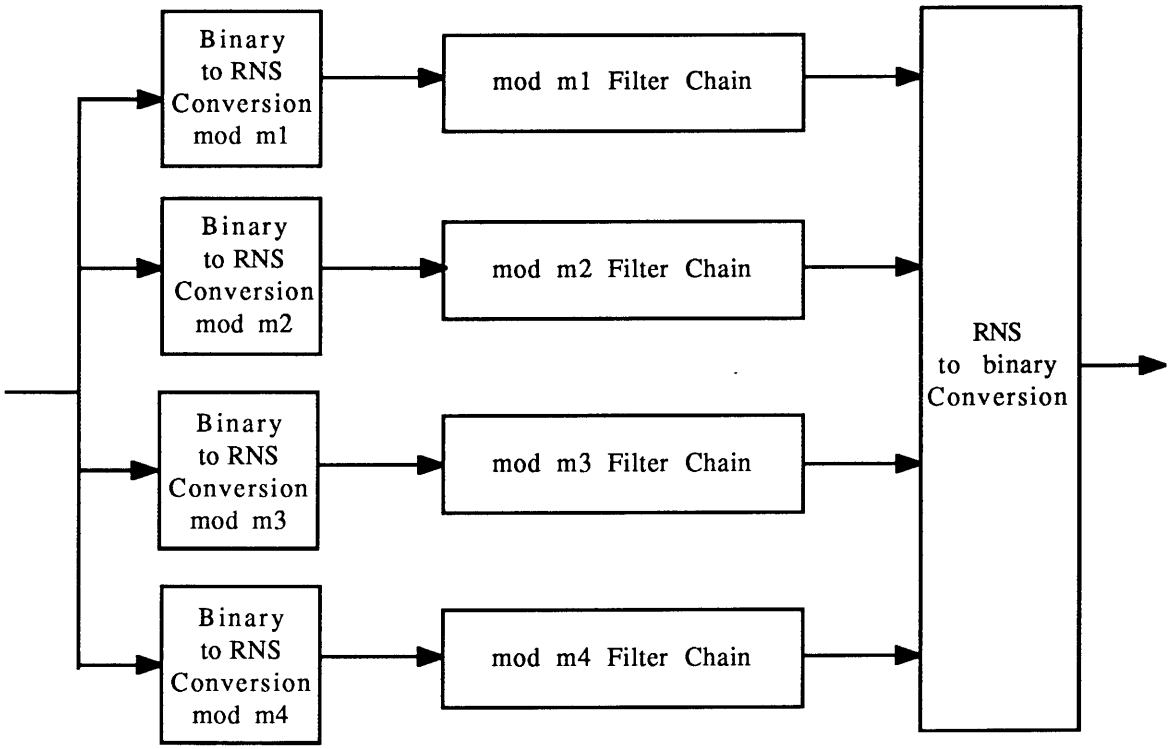


Figure 25 Residue FIR Filter System

The most significant restriction to the following architecture comparison is the interconnection constraint. A printed circuit card populated with SSI or MSI components can use several signal layers for the interconnection between discrete components; a typical VLSI process includes only two layers

of metal for interconnection between the custom blocks. As a result, interconnects in VLSI or WSI tend to consume area. A design that uses more gates but has simple interconnects, may be occupy less space than a design that has been optimized for gate count at the expense of a complex interconnection scheme.

A top level block diagram of a real residue FIR filter is shown in figure 25. The complete system should require only three basic designs: a binary to RNS conversion, a FIR filter tap, and an RNS to binary conversion. As a system level consideration, the binary to RNS conversion and FIR filter tap designs should be programmable to be used for any of the moduli in the system. The designs can be programmed to a particular modulus either by asserting a value to an input or by loading a value(s) into a register(s). This limits the number of specialized designs and makes the system expandable by adding an additional standard conversion block and RNS filter tap chain. Ideally, the RNS to binary conversion could be programmed to be used with any moduli set, but, because of the very structured computation that includes a set number of moduli for the system, it may be worthwhile to design the optimal moduli set into this component.

Residue FIR filter tap

The major focus of the design is the residue FIR filter tap. It is assumed that any filter being implemented with residue techniques will have a large number of taps to justify the added overhead of two conversion stages. As a result, the primary speed/size constraint will result from the filter chains. The filter tap will be discussed in depth in this section progressing from a simple approach to the more complex architectures. Two general design classes will be developed. One uses the biased/unbiased algorithm with normalized output that was developed for the residue function units in chapter 3. The other explores a slightly different approach of not normalizing the result of each tap, but instead, constraining the output to a limited range of values. Several filter tap designs will be presented and for each design type with hardware size and throughput estimated for each. The remaining two sections in this chapter will present conversion designs that are reasonable and match the throughput of the filter chains.

Brute Force

The obvious first approach to a residue FIR filter chain is to replace all of the arithmetic units in the transpose FIR filter block diagram with the corresponding residue arithmetic units. Each tap will need one residue adder, one residue multiplier, and a multiplier output fixup block. All have been developed in chapter 3. Although this design will work, a lot of hardware is required to construct each filter tap. Not including 2-1 MUXes, registers, or auxiliary gates, $2b^2 + b$ 1 bit adders are needed for each tap.

Coefficient Decomposition

Most of the complexity of the brute force approach comes from the residue multiplier. In each multiplier, b partial results (multiplies of individual coefficient bits with left shifted versions[1] of the input) are summed. If the partial results are not added together at each tap, but instead passed on to the next tap, the residue adds of the partial results need to be performed only once, at the end of the filter chain. In addition, because convolution is a linear operation, the left shifts of the input also only need to be performed once.

This approach was already examined in chapter 2 as a way to build a binary FIR filter without explicit multipliers. The same top level design can be used for an RNS filter by replacing the binary arithmetic units with their residue equivalents. At each subtap either the current input or 0 is added to the result of the previous stage. The subtaps are able to operate without multiplies because both the input and 0 are available without computation.

Looking closer, the basic idea exploited here is that the b individual bits of the coefficients select numbers (either 0 or current input) that are added to the b previous partial results. Now, if more versions of the current input are available such as: 1*input, 2*input, and 3*input, then the coefficient bits can be taken in groups of two to select between these three numbers and zero that could be added to the result of the prior stage. In this case the number of subtaps needed per tap is $\lceil b/2 \rceil$. In general, it is possible to represent the

---

[1] More exactly, versions of the input that have been recursively doubled. In the binary domain doubling is performed by a left shift; in the residue domain the result of the left shift must also be normalized.

coefficients in any fixed radix[1] integer number representation and precompute all multiples of the input that a single digit in this representation can span. For a base α decomposition of the coefficients $\lceil \log_\alpha \max(m) \rceil$ subtaps[2] are needed per tap.

To add some formalism to the development, it is useful to examine the mathematics involved. The coefficients are represented in base α notation as follows:

$$h[i] = \sum_{j=0}^{J-1} h_j[i]\, \alpha^j \qquad (1)$$

where $J = \lceil \log_\alpha \max(m) \rceil$ and $h_j[i]$ is the jth digit in the base α representation of h[i]. Inserting equation (1) into the convolution equation yields

$$y[n] = \sum_{i=0}^{N-1} \left[\sum_{j=0}^{J-1} h_j[i]\, \alpha^j \right] x[n-i]$$

By reversing the order of summation, two different computational procedures are generated.

$$y[n] = \sum_{j=0}^{J-1} \alpha^j \left[\sum_{i=0}^{N-1} h_j[i]\, x[n-i] \right] \qquad (2)$$

$$y[n] = \sum_{j=0}^{J-1} \sum_{i=0}^{N-1} h_j[i] \left[\alpha^j\, x[n-i] \right] \qquad (3)$$

The first procedure (eq 2) dictates that the mini-convolutions are computed with the results scaled by powers of α and then added. The second procedure (eq 3) dictates that the input is prescaled by powers of α and the results of the mini-convolutions are directly added. The difference between these two procedures was not significant in the binary case with $\alpha = 2$ because multiplying by factors of two in a binary representation is equivalent to left shifting. In the residue case, where computation must be performed to scale

---

[1] A fixed radix (base α) integer number system is defined by the following rule:

$$(\ldots a_3 a_2 a_1 a_0)_\alpha = \ldots + a_3\alpha^3 + a_2\alpha^2 + a_1\alpha^1 + a_0$$

where a_i are the digits of the radix α representation.

[2] Remember, m is the modulus of the channel, and therefore a range of distinct numbers is needed for the representation of the coefficients that meets or exceeds the maximum modulus.

by any number, the one procedure may be better than the other. This, however, will be addressed later when the scaling units are developed.

Base 2 - Bitwise

The simplest case of coefficient decomposition with the biased/unbiased algorithm is $\alpha = 2$, bitwise. The unbiased/biased architecture is identical to that shown in figure 8 with the binary subtaps in figure 9 replaced by residue conditional accumulators (figure 22). In the binary case the number of subtaps per tap was determined by the precision of the coefficients; in the residue case the number of subtaps is determined by the magnitude of the channel modulus (which effectively sets the precision of the coefficients within a particular channel). As shown above $\lceil \log_2 \max(m) \rceil \equiv b$ subtaps[1] are needed per channel. The complexity per tap for this design is b * b bit adders, or b^2 1 bit full adders which is less than one-half the number needed for the brute force design. Unfortunately, there is the added expense of broadcasting both the b bit current input and the b bit biased current input to all subtaps. A complete summary of hardware required per subtap is shown below

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b	9792b μ^2	32b
2-1 MUX	b	4896b+1632 μ^2	10b + 4
1 bit Register	b+2	8160b + 16320 μ^2	24b + 48
AND gate	b+1	4896b + 4896 μ^2	6b + 6
OR gate	1	4896 μ^2	6
Inverter	1	3264 μ^2	2
Totals	---	27744b + 31008 μ^2	72b + 66

Global Bus	2b signal lines
Critical Path	2-1 MUX + AND gate + b bit adder + OR gate + register
Throughput	$(1.9b + 8.06 \text{ ns})^{-1}$

Architecture Summary for base 2 subtap

[1] Because arithmetic is being performed using b bit binary arithmetic units, it seems logical to set the maximum modulus to that number which uses the full dynamic range of these units. In general, $\max(m) = 2^b$ where b is the chosen width of the channel.

Balanced Ternary

The advantage of going to a higher radix decomposition of the coefficients is that fewer subtaps are needed; the disadvantage is that more scaled versions of the input must be broadcast to each subtap. For example, if the coefficients are represented as standard radix 3 (ternary) with digits {0, 1, 2}, four b bit numbers must be broadcast to each subtap. Fortunately, there is no reason to use the standard digits. If the digits {-1, 0, 1} are used instead (balanced ternary), a simple trick permits a design that needs only two b bit numbers to be broadcast to each tap.

With the coefficients in a balanced ternary representation, there are four possible versions of the input ($\langle x \rangle_m$, $\langle x \rangle_m + \mu$, $\langle -x \rangle_m$, or $\langle -x \rangle_m + \mu$, selected by the coefficient and carry out of the prior stage) that could be added to the prior result at each subtap. However, in the balanced case the latter two can be easily derived from the former two. If a normalized unbiased residue in a b bit binary channel is two's complemented[1], the result is the biased version of the negative of the residue. If a normalized biased residue in a b bit binary channel is two's complemented, the result is the normalized version of the negative of the residue.[2]

$$2^b - \langle x \rangle_m = m + \mu - \langle x \rangle_m = \langle -x \rangle_m + \mu$$

$$2^b - (\langle x \rangle_m + \mu) = m + \mu - \langle x \rangle_m - \mu = \langle -x \rangle_m$$

Two's complementing can be built into hardware by using XOR gates to invert bits and using the carry in of the adder to perform the add 1. The final balanced ternary subtap using this technique is shown in figure 26. The coefficients are coded as follows:

h_1	h_0	ternary digit
0	0	0
0	1	1
1	1	-1
1	0	undefined

[1] Two's complement is a convenient way to represent both positive and negative numbers in a binary system. The two's complement of a b bit number x is obtained by subtracting x from 2^b, $-x = 2^b - x$. In the binary system this computation is equivalent to inverting each digit of x (0->1, 1->0) and adding 1 to the result.

[2] Remember when looking at the equations , $\mu = 2^b - m$, so $2^b = m + \mu$, and $\langle -x \rangle_m = m - \langle x \rangle_m$

Because three digits are being represented, two binary bits of register are necessary to hold the coefficient for a subtap. Since the coefficients are precalculated and can be placed in any unique digit/radix representation, the above coefficient definitions were chosen specifically to simplify the hardware decoding.

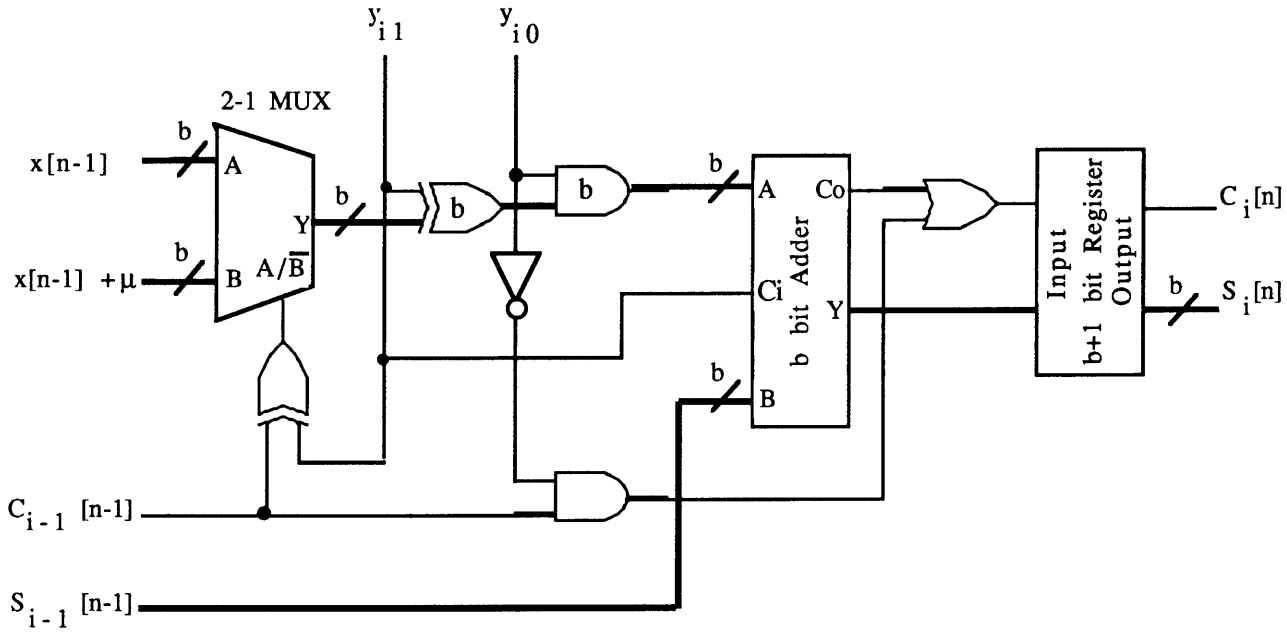


Figure 26 Balanced Ternary Subtap

At this point a logical concern is that the range of coefficients in a balanced tertiary system includes negative numbers. The range of numbers for a J digit balanced tertiary number system is $[-3^J/2 + 1, 3^J/2 - 1]$. Since m can be subtracted from any residue without changing its value, the negative numbers can be used. All that needs to be guaranteed is that the span of numbers in the coefficient range (3^J) is greater than or equal to the maximum modulus. The number of subtaps needed per tap is $\lceil \log_3 2^b \rceil$ where b is the number of bits in the binary channels. The chart below shows the number of subtaps needed for typical values of b.

b	$\lceil \log_3 2^b \rceil$
2	2
3	2
4	3
5	4
6	4
7	5
8	6
9	6

For b > 2, the balanced ternary representation of the coefficients does lower the number of subtaps needed per channel without increasing the number of global broadcast buses. This benefit is obtained at the expense of a slightly more complicated subtap, a two XOR gate increase in propagation delay, and a two bit coefficient at each subtap. The summary of the design is shown below

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b	9792b μ^2	32b
2-1 MUX + XOR	b	4896b + 3264 μ^2	16b + 6
1 bit Register	b+3	8160b + 24480 μ^2	24b + 72
AND gate	b+1	4896b + 4896 μ^2	6b + 6
XOR gate	1	4896 μ^2	10
OR gate	1	4896 μ^2	6
Inverter	1	3264 μ^2	2
Totals	---	27744b + 45696 μ^2	78b + 102

Global Bus	2b signal lines
Critical Path	XOR + (2-1 MUX + XOR) + AND + b bit adder + OR + register
Throughput	$(1.9b + 9.98\ ns)^{-1}$

Architecture Summary for balanced ternary subtap

Offset Radix 4

If we are willing to broadcast additional versions of the input we can achieve a radix 4 decomposition of the coefficients. As in the radix three case, the standard digit set {0, 1, 2, 3} is not optimal. For radix 4 one of the offset digit sets {-2, -1, 0, 1} or {-1, 0, 1, 2} should be used. For the design the former digit set will be (arbitrarily) chosen.

With the coefficients in the offset radix 4 representation, six versions of the input ($<-2x>_m$, $<-2x>_m + \mu$, $<-x>_m$, $<-x>_m + \mu$, $<x>_m$, or $<x>_m + \mu$, selected by the coefficient and carry out of the prior stage) could be added to the result of the prior stage. Using the two' s complementing trick, only four need to be broadcast to each subtap.

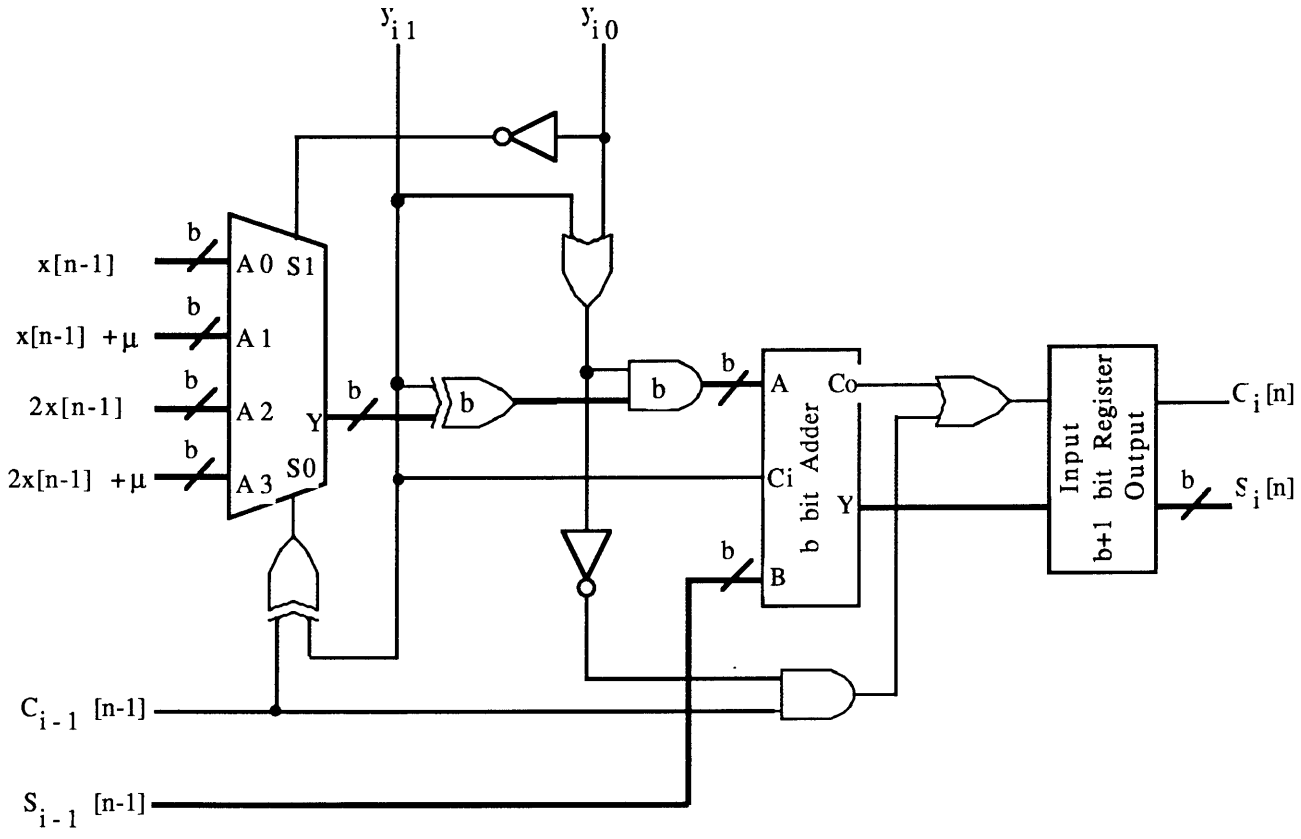


Figure 27 Offset Quaternary Subtap

The design is shown in a easy to understand positive logic form in figure 27. The coefficients are coded as follows:

h_1	h_0	quaternary digit
0	0	0
0	1	1
1	0	-2
1	1	-1

Using the same coefficient coding the design can be slightly optimized (two Inverters removed) by drawing the circuit in a less intuitive manner as shown in figure 28.

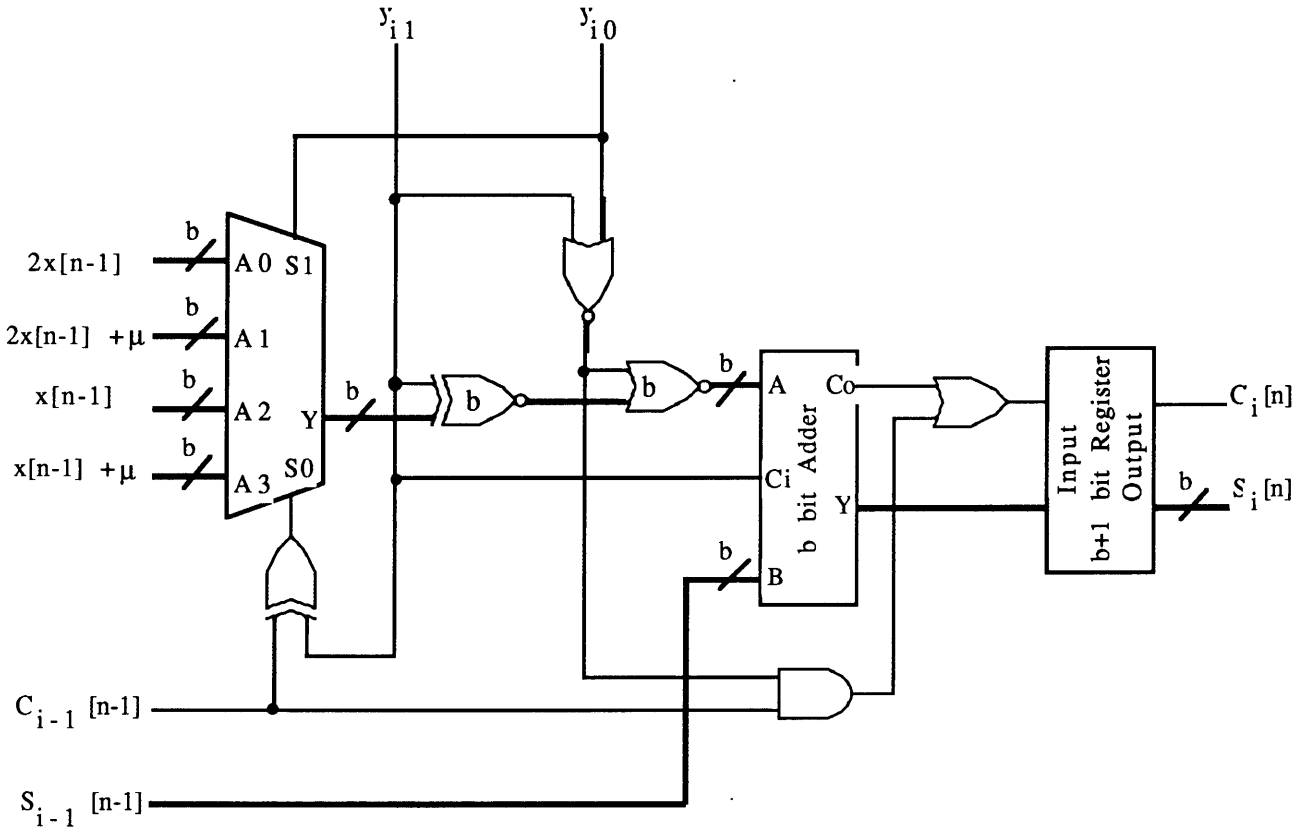


Figure 28 Offset Quaternary Subtap (Negative Logic)

The span of the coefficients in a J digit offset radix 4 representation is

$$\left[-\frac{2}{3}(4^J - 1),\ \frac{1}{3}(4^J - 1) \right]$$

The total span equals 4^J as it does in any radix 4 system. The corresponding number of subtaps per tap is $J = \lceil \log_4 2^b \rceil = \lceil b/2 \rceil$. The chart below lists J for some typical values of b.

b	$\lceil b/2 \rceil$
2	1
3	2
4	2
5	3
6	3
7	4
8	4
9	5

The number of subtaps per tap decreases from the number needed with the balanced ternary design. Unfortunately, the offset quaternary subtaps are more complicated, and the requisite number of global bus lines has doubled. The summary of the optimized hardware is shown below

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b	9792b μ^2	32b
4-1 MUX + XNOR	b	11424b + 11424 μ^2	26b + 34
1 bit Register	b+3	8160b + 24480 μ^2	24b + 72
NOR gate	b+1	4896b + 4896 μ^2	4b + 4
AND gate	1	4896 μ^2	6
XOR gate	1	4896 μ^2	10
OR gate	1	4896 μ^2	6
Totals	---	34272b + 55488 μ^2	86b + 134

Global Bus	4b signal lines
Critical Path	XOR + (4-1 MUX + XNOR) + NOR + b bit adder + OR + register
Throughput	$(1.9b + 12.14 \text{ ns})^{-1}$

Architecture Summary for the offset quaternary subtap

Balanced Quinary

The logical extension of the offset quaternary representation of the coefficients is a balanced quinary representation, radix 5 with digit set {-2, -1,

0, 1, 2}. No new innovations needed for the design which is shown in positive logic form in figure 29. The coefficients for the positive logic form are coded as follows:

h_2	h_1	h_0	quinary digit
0	0	0	0
0	0	1	1
0	1	1	2
1	0	1	-1
1	1	1	-2

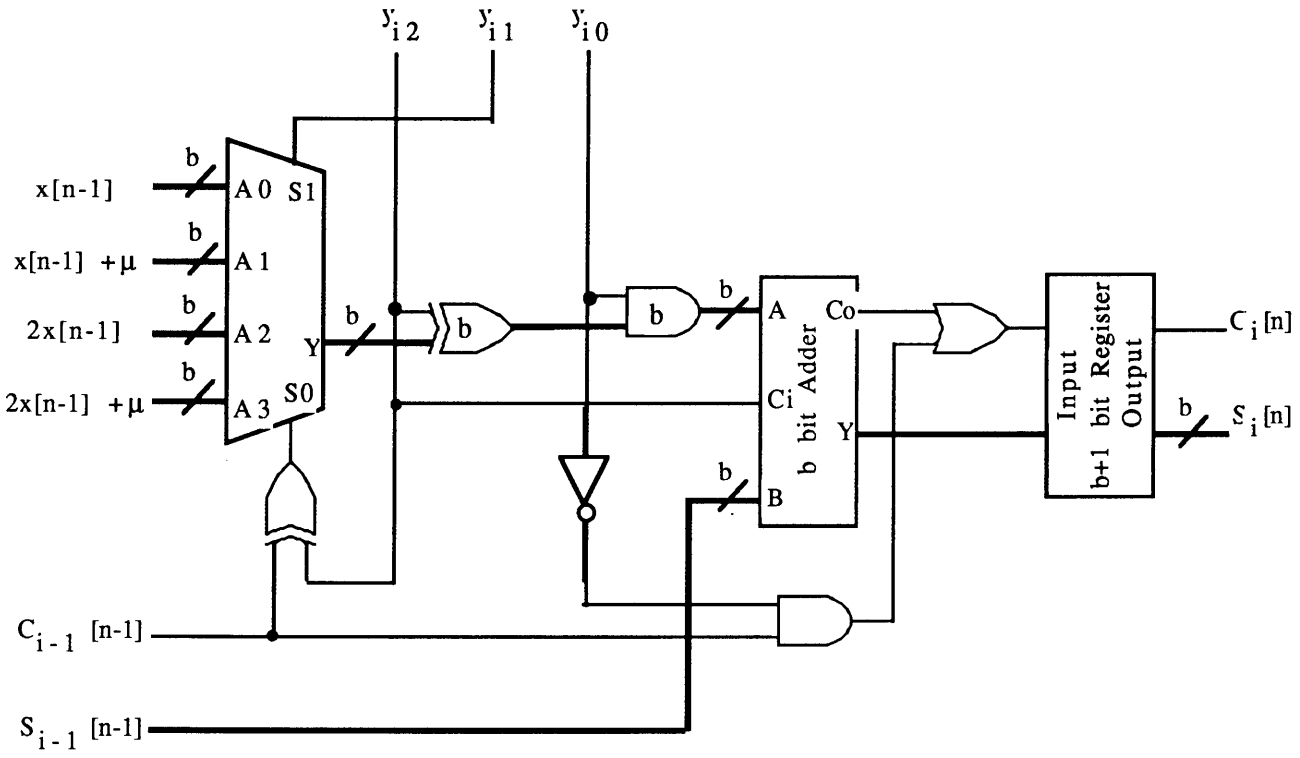


Figure 29 Balanced Quinary Subtap

By modifying the design slightly, it is possible to eliminate the inverter. The final version is shown in figure 30. This version requires a different coefficient coding as follows:

h_2	h_1	h_0	quinary digit
0	0	1	0
0	0	0	1
0	1	0	2
1	0	0	-1
1	1	0	-2

The only difference between the two coefficient sets is that h_0 has been inverted.

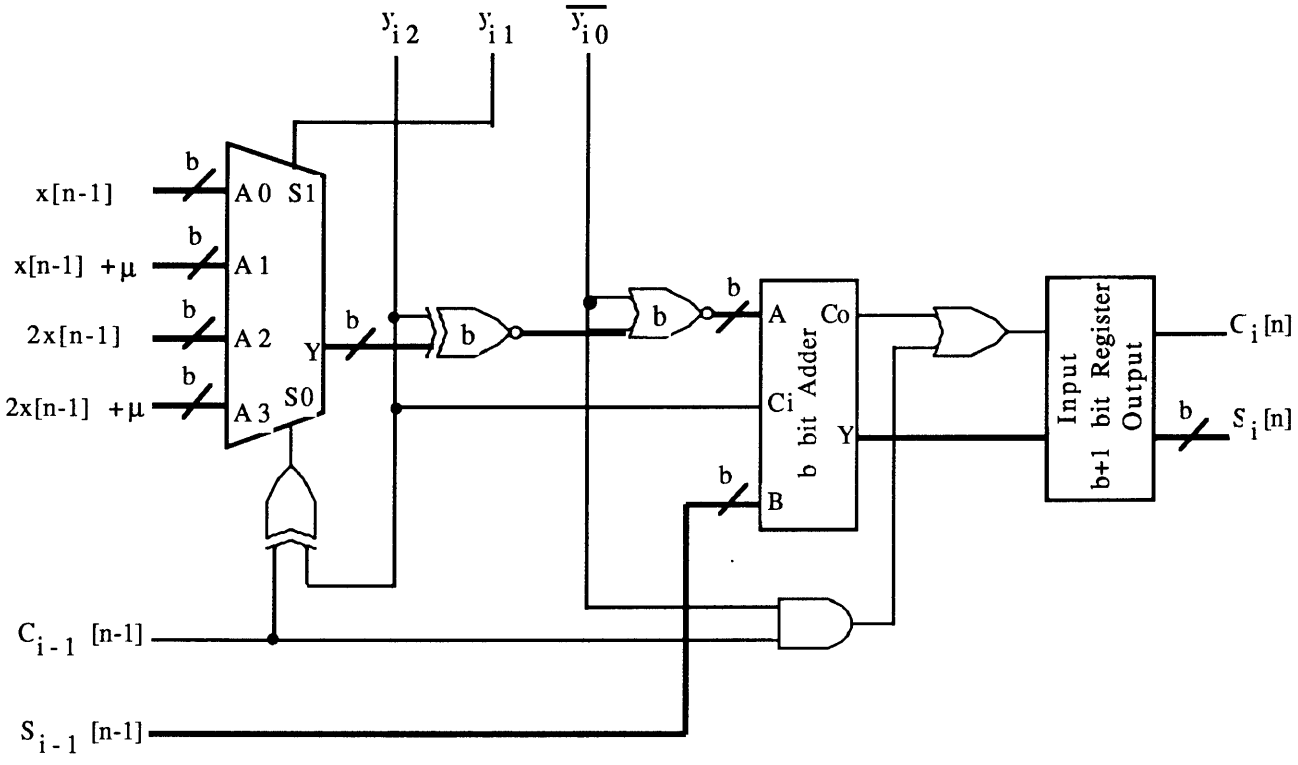


Figure 30 Balanced Quinary Subtap (negative logic)

The span of the coefficients in a J digit balanced radix 5 representation is

$$\left[-\frac{1}{2}5^J , \frac{1}{2}5^J \right]$$

The total span equals 5^J as it does in any radix 5 system. The corresponding number of subtaps per tap is $J = \lceil \log_5 2^b \rceil$. The chart below lists J for some typical values of b.

b	$\lceil \log_5 2^b \rceil$
2	1
3	2
4	2
5	3
6	3
7	4
8	4
9	4

Unfortunately, the advantage of using balanced quinary is not realized until b is greater than or equal to 9. Although the balanced quinary subtap requires the same number of global buses and has the same throughput as the offset quaternary design, the marginally extra hardware would not be warranted unless $b \geq 9$.[1] For those truly massive dynamic range requirements, the architecture summary is shown below

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b	9792b μ^2	32b
4-1 MUX + XNOR	b	11424b + 11424 μ^2	26b + 34
1 bit Register	b+4	8160b + 32640 μ^2	24b + 96
NOR gate	b	4896b μ^2	4b
AND gate	1	4896 μ^2	6
XOR gate	1	4896 μ^2	10
OR gate	1	4896 μ^2	6
Totals	---	34272b + 58752 μ^2	86b + 152

Global Bus	4b signal lines
Critical Path	XOR + (4-1 MUX + XNOR) + NOR + b bit adder + OR + register
Throughput	$(1.9b + 12.14 \text{ ns})^{-1}$

Architecture Summary for the offset quinary subtap

[1] Proof that aesthetics and symmetry are not the only things that is important

Subtap Summary

The coefficients could be decomposed into even higher radix representations, but the disadvantages of having more global buses and larger multiplexors would outweigh any advantages that would be obtained by having fewer subtaps. At this point it is most instructive to examine the four designs for different values of b. The tables below list the size and number of transistors needed per tap for each of the four designs. The values were calculated using the numbers from the architecture summary for each design.

		per subtap		entire tap	
b	J	size (μ^2)	transistors	size (μ^2)	transistors
2	2	86496	210	172992	420
3	3	114240	282	342720	846
4	4	141984	354	567936	1416
5	5	169728	426	848640	2130
6	6	197472	498	1184832	2988
7	7	225216	570	1576512	3990
8	8	252960	642	2023680	5136
9	9	280704	714	2526336	6426

Binary

		per subtap		entire tap	
b	J	size (μ^2)	transistors	size (μ^2)	transistors
2	2	101184	258	202368	516
3	2	128928	336	257856	672
4	3	156672	414	470016	1242
5	4	184416	492	737664	1968
6	4	212160	570	848640	2280
7	5	239904	648	1199520	3240
8	6	267648	726	1605888	4356
9	6	295392	804	1772352	4824

Balanced Ternary

		per subtap		entire tap	
b	J	size (μ^2)	transistors	size (μ^2)	transistors
2	1	124032	306	124032	306
3	2	158304	392	316608	784
4	2	192576	478	385152	956
5	3	226848	564	680544	1692
6	3	261120	650	783360	1950
7	4	295392	736	1181568	2944
8	4	329664	822	1318656	3288
9	5	363936	908	1819680	4540

Offset Quaternary

		per subtap		entire tap	
b	J	size (μ^2)	transistors	size (μ^2)	transistors
2	1	127296	324	127296	324
3	2	161568	410	323136	820
4	2	195840	496	391680	992
5	3	230112	582	690336	1746
6	3	264384	668	793152	2004
7	4	298656	754	1194624	3016
8	4	332928	840	1331712	3360
9	4	367200	926	1468800	3704

Balanced Quinary

Figures 31 and 32 provide a graphic comparison of the size and number of transistors, respectfully, for the different designs. Although the balanced base 3 appears to be marginally better for b = 3 and the balanced base 5 appears to be clearly better for b = 9, the offset base four design seems to have an advantage for all values of b in between. Assuming a large number of filter taps, it is possible to neglect the scaling and summing units needed for the different designs.[1] The differences between the global busing requirements, however, can not be neglected. Both the binary and the balanced ternary implementations need 2b global bus lines[2]; the offset quaternary and the balanced quinary need 4b lines.

---

[1] Remember, the output of each mini-convolution (subtap) chain must be scaled and summed with the scaled outputs of the other chains.

[2] In reality, the clock must be globally broadcast also, but this is common to all designs.

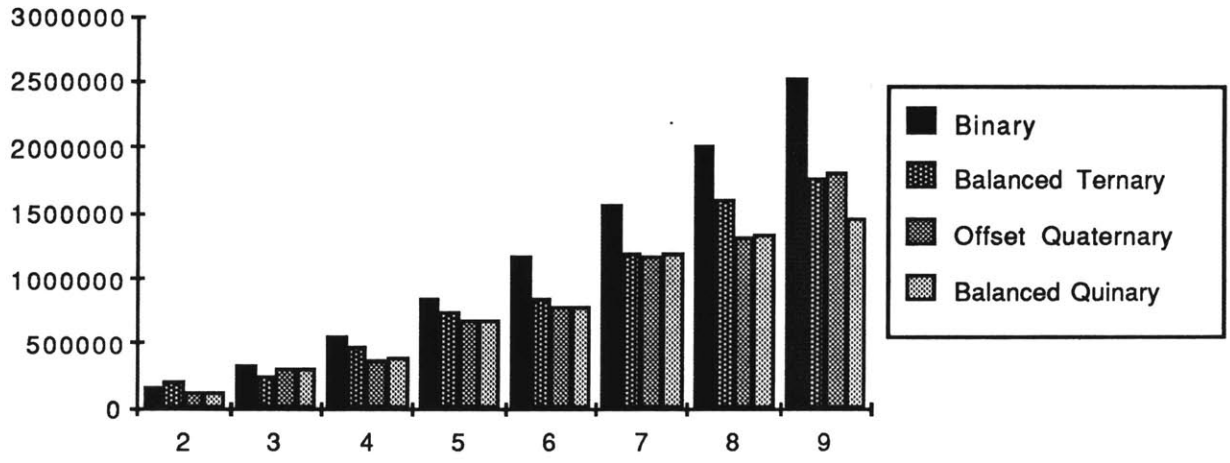


Figure 31 Size (μ^2) verses Number of Bits in Binary Channel

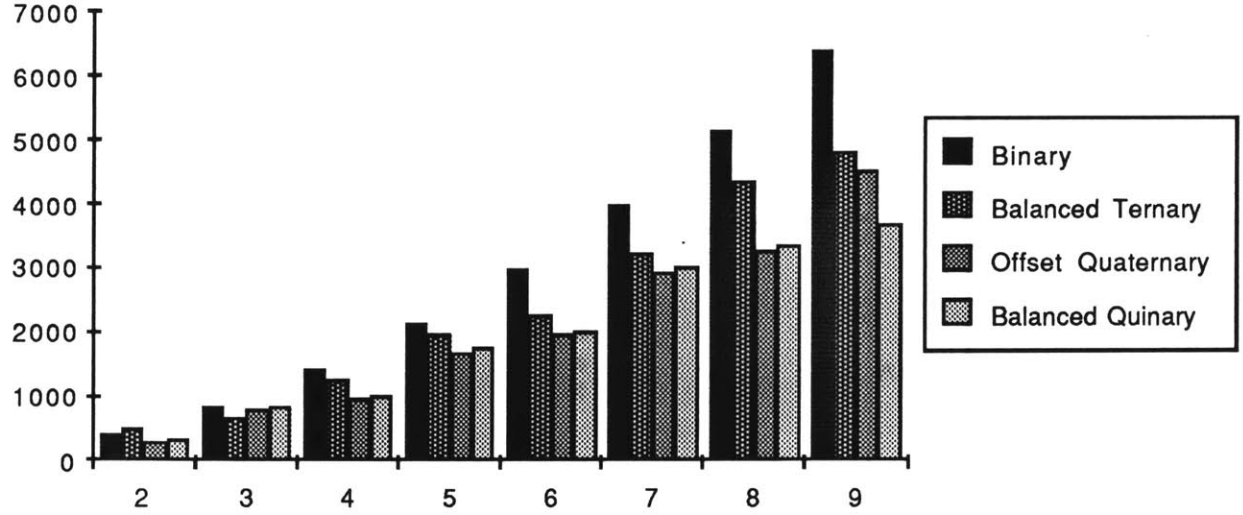


Figure 32 Transistors verses Number of Bits in Binary Channel

The delay through the subtaps increases monotonically as the radix increases or the number of bits in the binary channel increases. A summary of the latency[1] through a single subtap is shown in the following table:

[1] As are the hardware sizing estimates, the latency estimates are derived from a standard cell library. See the appendix...

b	binary	balanced ternary	offset quaternary	balanced quinary
2	11.86 ns	13.78 ns	15.94 ns	15.94 ns
3	13.76 ns	15.68 ns	17.84 ns	17.84 ns
4	15.66 ns	17.58 ns	19.74 ns	19.74 ns
5	17.56 ns	19.48 ns	21.64 ns	21.64 ns
6	19.46 ns	21.38 ns	23.54 ns	23.54 ns
7	21.36 ns	23.28 ns	25.44 ns	25.44 ns
8	23.26 ns	25.18 ns	27.34 ns	27.34 ns
9	25.16 ns	27.08 ns	29.24 ns	29.24 ns

Scaling and Summing

Although the scaling and summing operations can be neglected for the hardware comparison between the different coefficient decomposition designs, it is an essential part of the complete design.

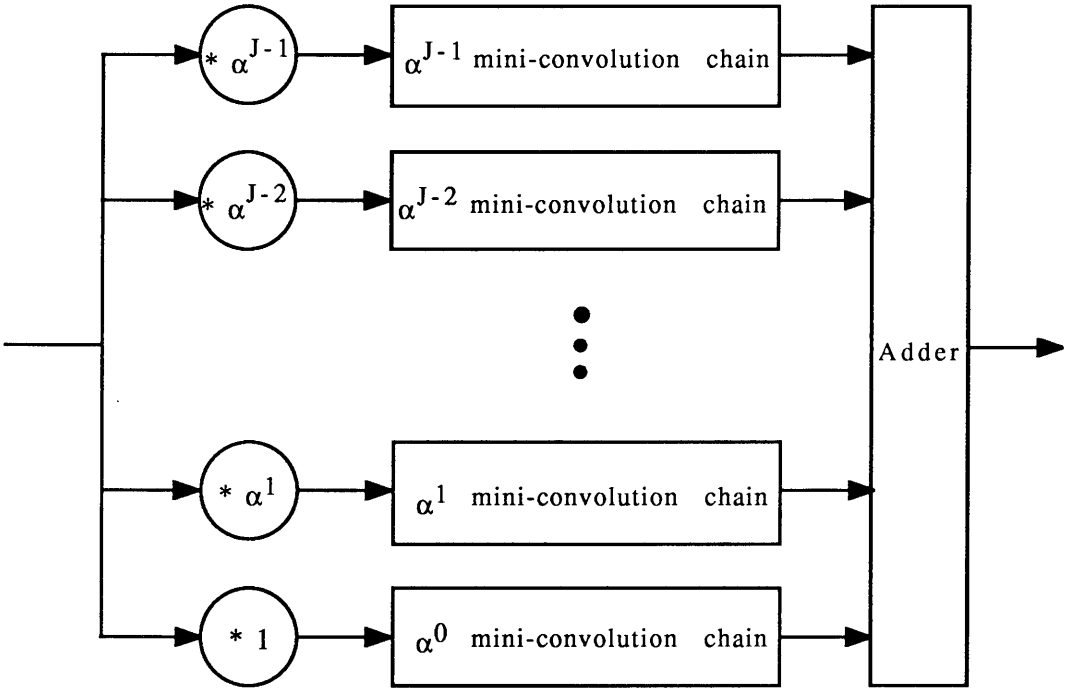


Figure 33 Premultiplication by Powers of the Radix

Earlier in this chapter two computational procedures were discussed. Either the input can be premultiplied by powers of the radix (figure 33), or the results of the mini-convolution chains can be postmultiplied by powers of the radix as shown in (figure 34). To simplify the computation and decrease

the necessary hardware, a form of Horner's Algorithm[1] can be used as seen in figures 35 and 36. The use of Horner's Algorithm has the complication of requiring that the data arrival times be skewed in the mini-convolution channels[2] so that the inputs to the final adder chain arrive at the proper times. To minimize the number of registers needed to skew the data, the first computational procedure is used in the form shown in figure 35. The latency for the multiply by α blocks equalizes the delay needed between outputs of consecutive subtaps. If the second computational procedure is used, a chain of registers would be needed to provide delayed versions of the input to each mini-convolution chain.

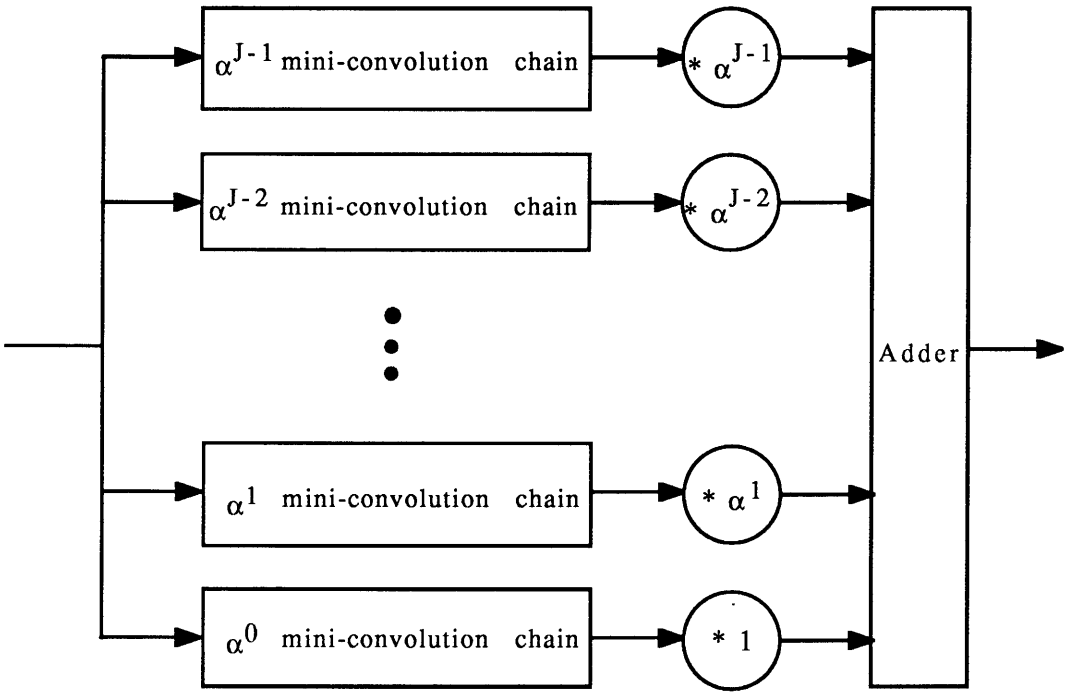


Figure 34 Postmultiplication by Powers of the Radix

With a computational procedure finally chosen, scaling units must be developed to multiply by the radices {2, 3, 4, 5} used in the designs. Obviously, the minimum latency designs are desired, but the scaling blocks must operate with a throughput equal to that of the filter subtaps or, equivalently, equal to

---

[1] Horner's Algorithm is usually associated with polynomial evaluation where $\sum_{i=0}^{n} a_i x^i$ is evaluated as $a_0 + x(a_1 + x(a_2 + x(a_3 + x(\ \ldots \ ))))$

[2] The ith subtap in each mini-convolution chain will be operating on data from different input times.

one clock cycle. A multiply by 2 block has already been developed (figure 20) that meets the timing requirements. It has a throughput and latency[1] equal to one clock cycle and outputs both biased and unbiased forms of the product. The multiply by 4 block would consist of two consecutive multiply by two blocks. It would have a throughput of one clock cycle, but a latency of two (three) clock cycles. The *4 block would also output both biased and unbiased forms of the product.

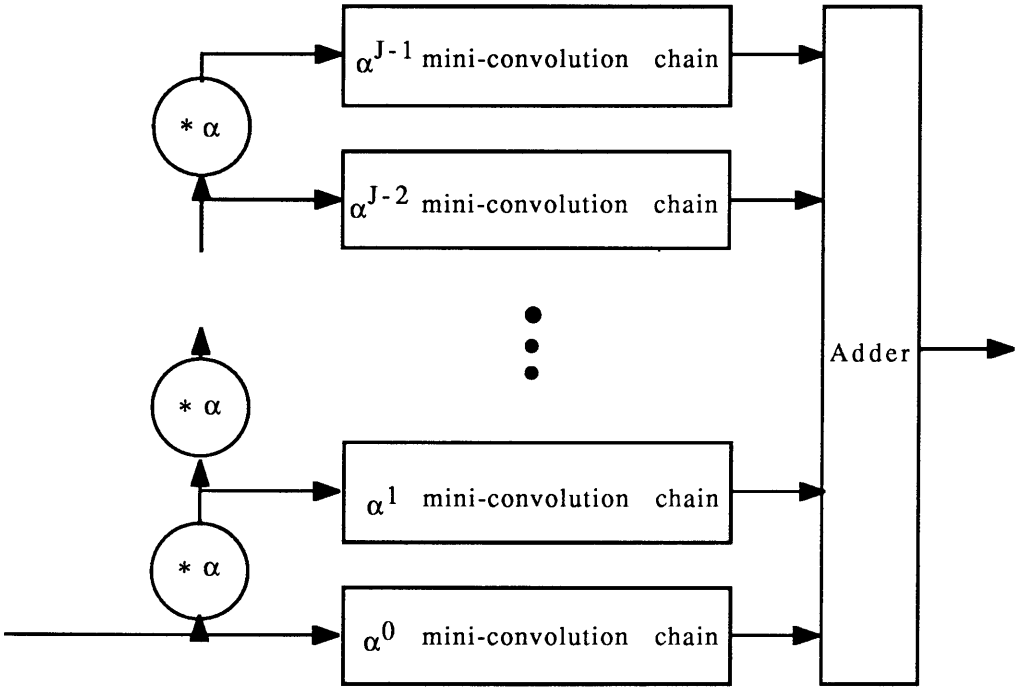


Figure 35 Horner's Algorithm for Premultiplication

Unfortunately, a multiply by 3 unit is not as simple as the previous two. A combination of one-half of a *2 unit with additional hardware is needed to multiply by 3. A design with a latency of two is shown in figure 38. The number of adders could be reduced by adding x to 2x and conditionally adding μ to the result; cost of this hardware optimization, however, is an increase in latency to 3 clock cycles.

---

[1] Assuming both x and x+m are available. Otherwise, an extra stage would be necessary to generate both unbiased and biased versions of the multiplicand. This extra stage would have a throughput of one clock cycle, but would increase the total latency to two clock cycles. One half of the multiply by two block will operate on an unbiased input to produce only an unbiased result in one clock cycle.

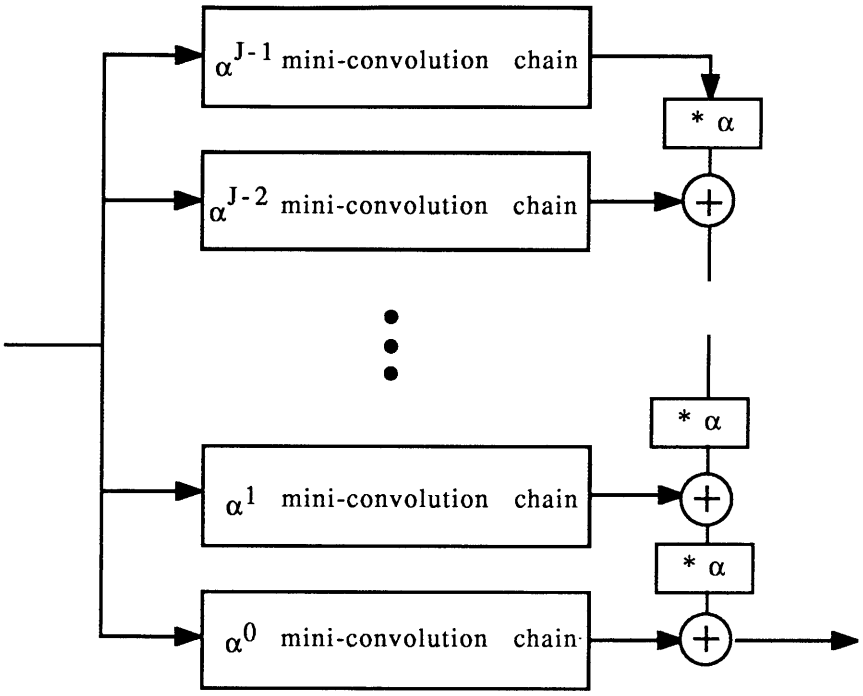


Figure 36 Horner's Algorithm for Postmultiplication

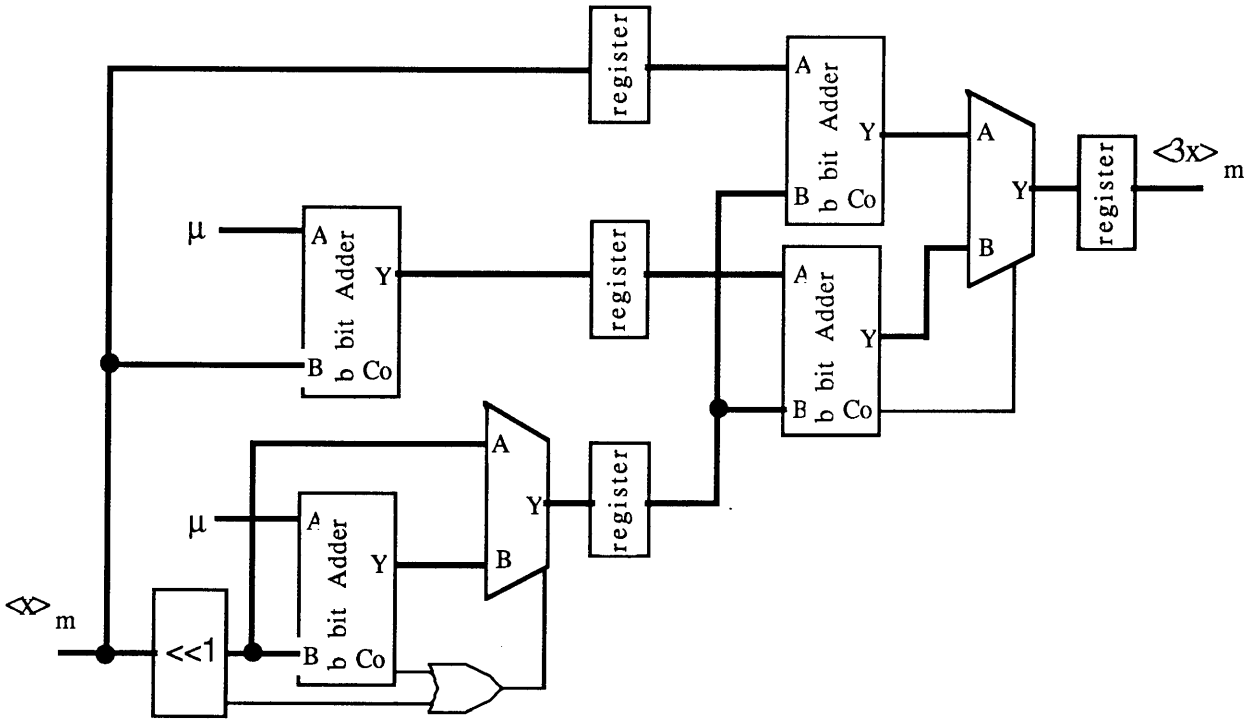


Figure 37 Multiply by 3 Block

The multiply by 5 unit is very similar to the multiply by three block. A design with a latency of three is shown in can be implemented by adding an additional multiply by 2 block to the *3 design and an additional delay register for the input. As with the *3 block, the number of adders could be reduced by adding x to 4x and conditionally adding μ to the result at the cost of increased latency.

New Algorithm with Fewer Buses

To reduce the number of globally broadcast signals needed for larger radix decompositions, it is necessary to back up and examine the basis of the biased/unbiased designs in more depth. In each design the number of bits used in the binary channels is equal to the maximum number of bits that a normalized residue could occupy. The dynamic range restriction forces the output of each subtap to be normalized[1] because unnormalized numbers could be uniquely represented. The normalization is achieved by guaranteeing that a normalized unbiased residue is always added to a normalized biased one. The output of the previous stage will always be in normalized[2] but may or may not be biased. Because of the uncertain state of the previous stage's output, both unbiased and biased versions of each normalized digit multiple of the input must be precalculated and bused to the subtaps. This clever procedure used to keep residues normalized within a binary channel has led to the large number of buses.

The Algorithm

If more bits are included in the binary channels than are needed to represent the normalized residues, unnormalized residues can also be uniquely represented. Instead of requiring that the output of a subtap be normalized, the output can be restricted to some range of values modulo m. At each subtap, add in the new product plus positive or negative multiples of m as required to keep the result within some restricted range. With a sufficient number of bits in the binary channels, the sum of two unbiased residues,

---

[1] Generally, normalized residue implies that the magnitude of the residue falls within the range [0, m-1]. In this case, because the dynamic range of the channel is equal to the dynamic range need for the maximum modulus, the "normalized range" could actually be any offset of the standard normalized range (ie $[a, m+a-1] \mid a \in Z$). The point is that the span of the output cannot exceed m uniquely.

[2] Inductive reasoning

either of which may not be normalized, will result in an unbiased residue, which also may not be normalized. Since the output of the previous stage will always be unbiased[1], we only need to provide the subtaps with unbiased digit multiples of the input. If these digits are powers of two, the unbiased digit multiples of the coefficient can be calculated at the subtap by a left shifter. Negative values can be calculated by two's complementing.

To support negative values in an unbiased environment, the two's complement representation of negative binary numbers must be used. In the previous designs all numbers were considered positive. The carry out of the adder merely indicated whether the sum was biased or unbiased. Even the two's complementing trick used to invert residues generated positive values[2]. However, with true negative numbers in the system a method exists to ensure that the output of a subtap lies within a certain range.

In order to keep the temporary results form growing in magnitude, multiples of m must be added or subtracted from the accumulation. By adding and subtracting multiples of both x (the current input) and x - m (now a true negative number), the multiples of m can be automatically added or subtracted to keep the result within a specified range. If the previous result is negative, a positive number is added to it; if the previous result is positive, a negative number is added to it. The subtap algorithm is listed below with the following notation[3]: $p_i[n]$ equal to the result of the ith stage at the nth time step and $h_j[N-i]$ equal to the jth digit in the balanced radix α decomposition of the N-i th coefficient.

```
if (hj[N-i] == 0)                                   /* case 0 */
        pi[n] = pi-1[n-1]

if (hj[N-i] > 0  &&  pi-1[n-1] ≥ 0)                 /* case 1 */
        pi[n] = pi-1[n-1]  +  hj[N-i]  *  (x-m)

if (hj[N-i] > 0  &&  pi-1[n-1] < 0)                 /* case 2 */
        pi[n] = pi-1[n-1]  +  hj[N-i]  *  x
```

---

[1] Inductive reasoning, again

[2] When $\langle x \rangle_m$ was inverted to generate $\langle -x \rangle_m$, the result $\langle -x \rangle_m$ equaled m - $\langle x \rangle_m$ not - $\langle x \rangle_m$.

[3] Some of this notation was developed in section 2.2.2 in the discussion of the transpose filter

```
if (hj[N-i] < 0  &&  pi-1[n-1] ≥ 0)                    /* case 3 */
        pi[n] = pi-1[n-1]  +  hj[N-i] * x

if (hj[N-i] < 0  &&  pi-1[n-1] < 0)                    /* case 4 */
        pi[n] = pi-1[n-1]  +  hj[N-i] * (x-m)
```

Examining the magnitudes of the quantities involved in the above algorithm shows the range that the output, $p_i[n]$, can span and therefore the number of bits needed in the binary channels. The magnitude of $p_i[n]$ will be the largest when $p_i[n-1]$ is close to zero and the magnitude of the coefficient hj[N-i] equals its maximum. Also, because the x spans from 0 to m-1 and x-m spans from -m to -1, it is expected that the cases including x-m would contribute to the largest results. Regardless, each case in the algorithm will be examined separately. For case #1, since a negative number is being added to $p_{i-1}[n-1]$, the maximum magnitude of $p_i[n]$ will equal $-m*\max(h_j)$ when $p_{i-1}[n-1] = 0$. For case #2, since a positive number is being added to $p_{i-1}[n-1]$, the maximum magnitude of $p_i[n]$ will equal $(m-1)*\max(h_j) - 1$ when $p_{i-1}[n-1] = -1$. For case #3, the maximum magnitude of $p_i[n]$ will equal $(m-1)*\max(h_j)$. Finally, for case #4, the maximum magnitude will equal $-m*\min(h_j) - 1$. Collecting these results, the output spans the range $-m*\max(h_j)$ to $-m*\min(h_j) - 1$. If h[n] is decomposed in a balanced radix system with all digits equal to powers of two and the maximum value of m equal to 2^b, the span of the output can be efficiently[1] represented in a b+c+1 bit two's complement binary channel where $c = \log_2 (\max(h_j))$ and an extra bit is used for the sign.

Because the temporary values in each mini-convolution chain can span the range $\pm$ m* max(h_i), some method is needed to normalize the values after the final tap. Although the calculation which consists of adding/subtracting multiples of m will not require a significant amount of hardware, it is required and therefore must be considered. Also, as is part of the original algorithm, the outputs of the final subtaps must be scaled and summed to form the final result for the residue channel. Again, for a large number of taps this hardware will not be significant, but still must be incorporated into the

---

[1] Efficient implies unique representation with no wasted dynamic range for the case $m = 2^b$

complete system. Initially, these two components of the system will be ignored. After the different subtap designs have been completed, we will return to these designs.

The Hardware

The three implementations that will be examined are balanced ternary, balanced quinary, and balanced septary (base 7). The hardware required to implement the algorithm in the previous section is very similar to the design of the coefficient decomposition subtaps. The major differences are that the binary channels are slightly wider, and some method is needed to perform left shifts. A general block diagram of the component elements is shown in figure 38.

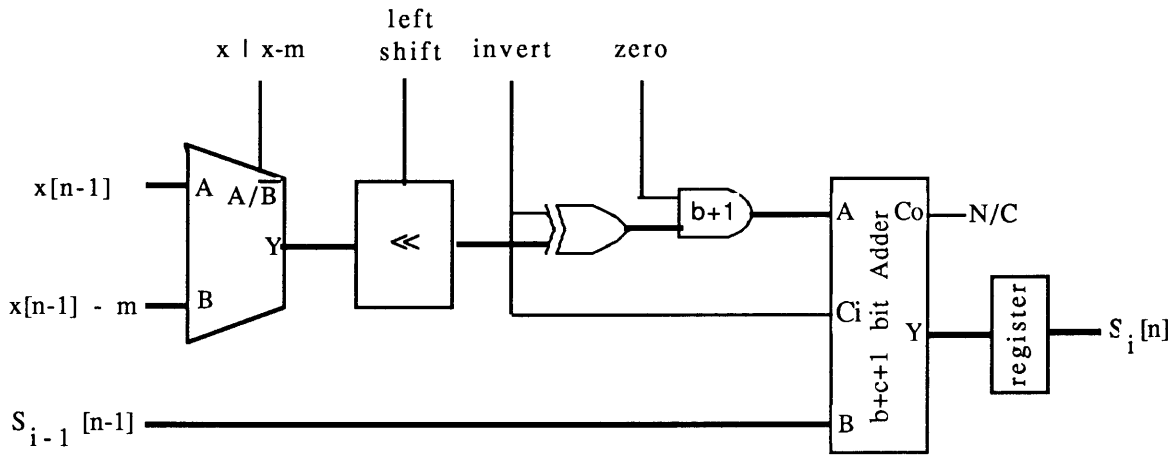


Figure 38 New Algorithm Subtap Architecture

Balanced Ternary

The first implementation, balanced ternary, does not need the left shift capability because the only digits are -1, 0, and 1. The basic design of subtap is shown in figure 39. It is virtually the same as the coefficient decomposition subtap except that no carry forwarding circuitry is needed. The coefficients are decoded as follows:

h_1	h_0	ternary digit
0	0	0
0	1	1
1	1	-1
1	0	undefined

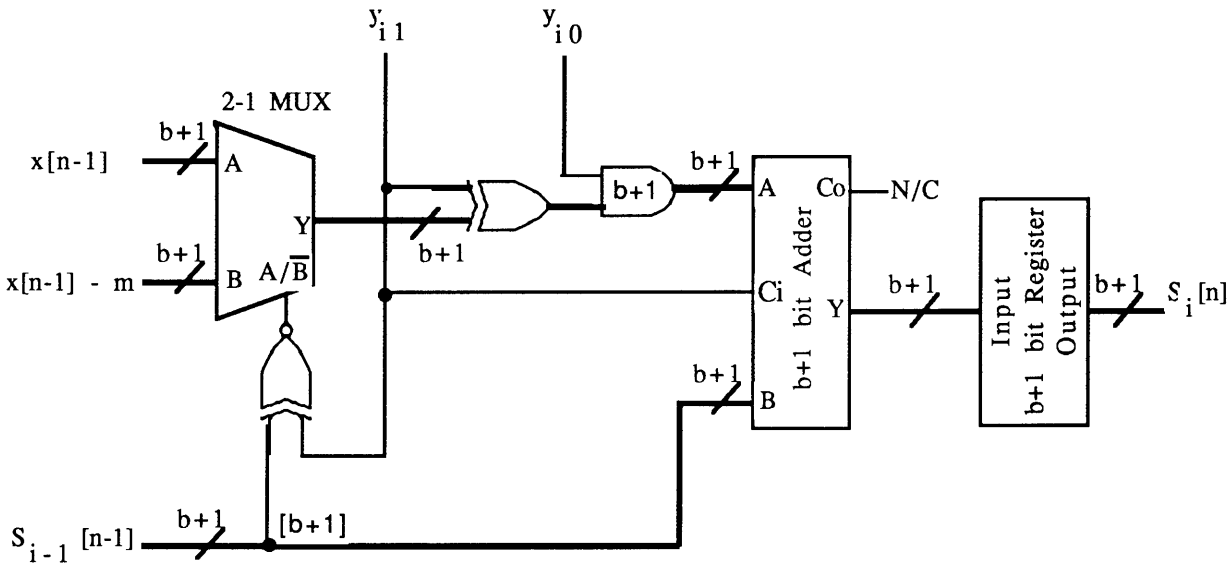


Figure 39 New Balanced Ternary Subtap

The high order bit of the previous result indicates whether the previous result is negative, and the high order bit of the coefficient indicates whether the coefficient is negative. The XNOR combination of these signals assures that a positive number is always added to a negative one and a negative always added to a positive.

The design can be slightly optimized by eliminating one stage of the 2-1 MUX. Because the maximum value of m is 2^b, the maximum value of x is 2^b - 1 and the minimum value of x-m is -2^b. So, the high order bit of x is always set to 0, and the high order bit of x-m is always set to 1. Because the output of the XNOR is 0 to select x and 1 to select x-m, it can provide the high order bit of x and x-m. In addition, the high order bits of x and x-m do not need to be globally broadcast. The final design is shown in figure 40 .

Although the carry forwarding circuitry has been removed and with it an OR gate in the critical path, the overall design is probably worse than the

corresponding unbiased/biased design. More gates were added than were removed, and the propagation delay of additional carry stage in the adder is

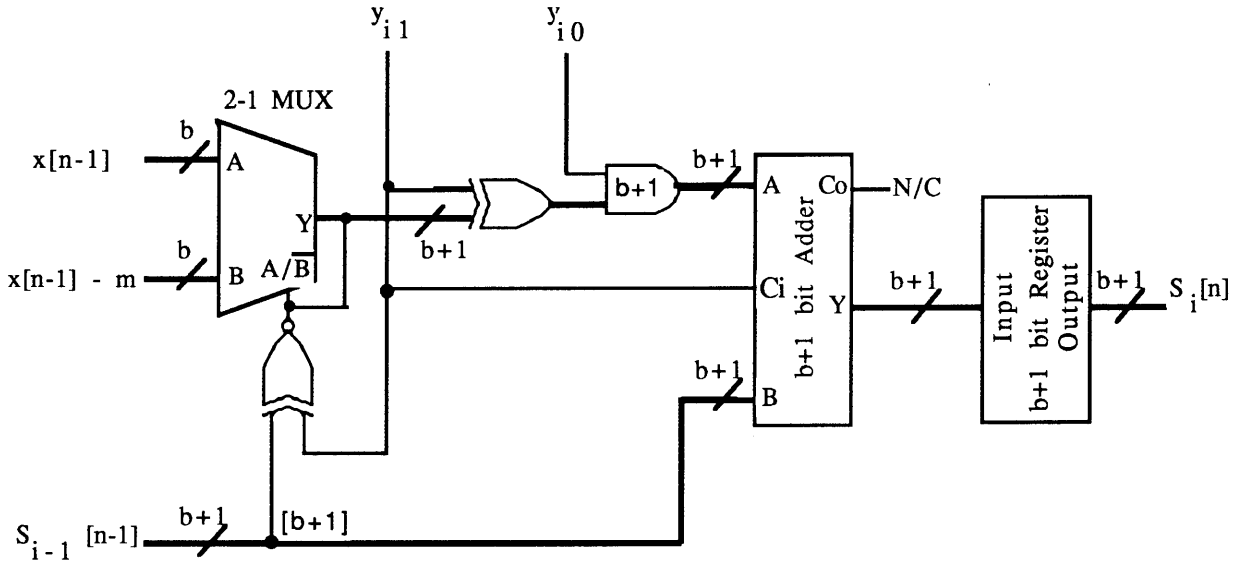


Figure 40 Improved New Balanced Ternary Subtap

longer than the delay of the removed OR gate. Also, the same number of bus lines are needed. At least this design shows proof of concept. The architecture summary is listed below

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b+1	9792b + 9792 μ^2	32b + 32
2-1 MUX + XOR	b	4896b + 3264 μ^2	16b + 6
1 bit Register	b+3	8160b + 24480 μ^2	24b + 72
AND gate	b+1	4896b + 4896 μ^2	6b + 6
XOR gate	1	4896 μ^2	10
XNOR gate	1	4896 μ^2	8
Totals	---	27744b + 52224 μ^2	78b + 134

Global Bus	2b signal lines
Critical Path	XNOR + (2-1 MUX + XOR) + AND + b+1 bit adder + register
Throughput	$(1.9b + 10.36\ ns)^{-1}$

Architecture Summary for balanced ternary subtap #2

Balanced Quinary

The purpose of the subtap algorithm is to lower the number of global buses, not to reduce the amount of hardware per subtap. Because the subtap algorithm requires two numbers to be globally broadcast, the balanced ternary implementation could not have been expected to be an improvement. The new balanced quinary implementation, however, will reduce the number of globally broadcast numbers from four to two.

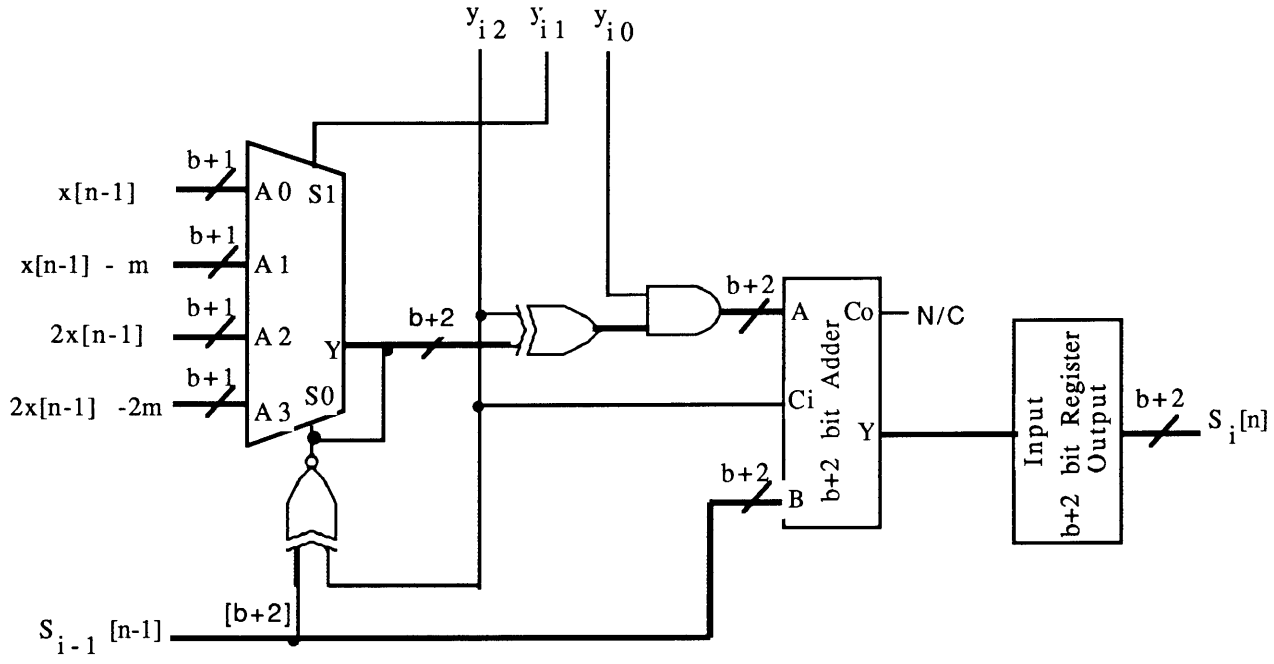


Figure 41 New Balanced Quinary Subtap

The basic design will be very similar to the balanced ternary design except that hardware must be added to perform left shifts. Because the standard cell library that I am using does not include a left shifter, a b + 1 bit MUX with hardwired left shifted versions x and x-m is used. The b+1th bit of the x input to the MUX is tied to 0, the b+1th bit of the x-m input is tied to 1; the low order bits of both 2x and 2x-2M inputs is tied to 0. The high order bit (b+2 bit) of all inputs is added by the selector line after the MUX. The final design is shown in **figure 43**. The coefficients are decoded as follows:

h_2	h_1	h_0	quinary digit
0	0	0	0
0	0	1	1
0	1	1	2
1	0	1	-1
1	1	1	-2

Once again, the subtap algorithm results in more hardware and longer propagation delays. However, the number of globally broadcast lines has been reduced from 4b to 2b. At this point a custom VLSI layout of both balanced quinary designs is necessary to determine the relative hardware cost of an additional bus line. Also, the left shifter could be implemented as a partially populated grid of transmission gates[1], and a b bit 2-1 MUX could be used for to choose between x and x-m. Regardless, within the available technology, the architecture summary is as follows:

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b+2	9792b + 19584 μ^2	32b + 64
4-1 MUX + XOR	b+1	8160b + 19584 μ^2	24b + 58
1 bit Register	b+5	8160b + 40800 μ^2	24b + 120
AND gate	b+2	4896b μ^2	6b + 12
XOR gate	1	4896 μ^2	10
XNOR gate	1	4896 μ^2	8
Totals	---	31008b + 89760 μ^2	86b + 272

Global Bus	2b signal lines
Critical Path	XNOR + (4-1 MUX + XOR) + AND + b+2 bit adder + register
Throughput	$(1.9b + 14.06)^{-1}$

Architecture Summary for the balanced quinary subtap #2

[1] The grid would be rectangular with transmission gates on the two central diagonals to allow the rows (input) to be directly passed to the columns (output) or to allow the rows to be passed shifted left one place. Once the select lines have been set, the propagation delay through the shifter would only be one transmission gate delay. A similar design with a fully populated grid is frequently used as a barrel shifter.

Modified Balanced Septary

Because the new subtap algorithm only requires two b bit numbers to be broadcast, a radix seven design can be implemented. With the old biased/unbiased algorithm, six b bit numbers would have been needed. The standard balanced septary digits would be the set {-3, -2, -1, 0, 1, 2, 3}. Because we are trying to avoid real multiplications, the modified digit set {-4, -2, -1, 0, 1, 2, 4} will be used. Although the modified digit set allows all multiplications to be performed by left shifts, it does put some restrictions on the range of numbers spanned.

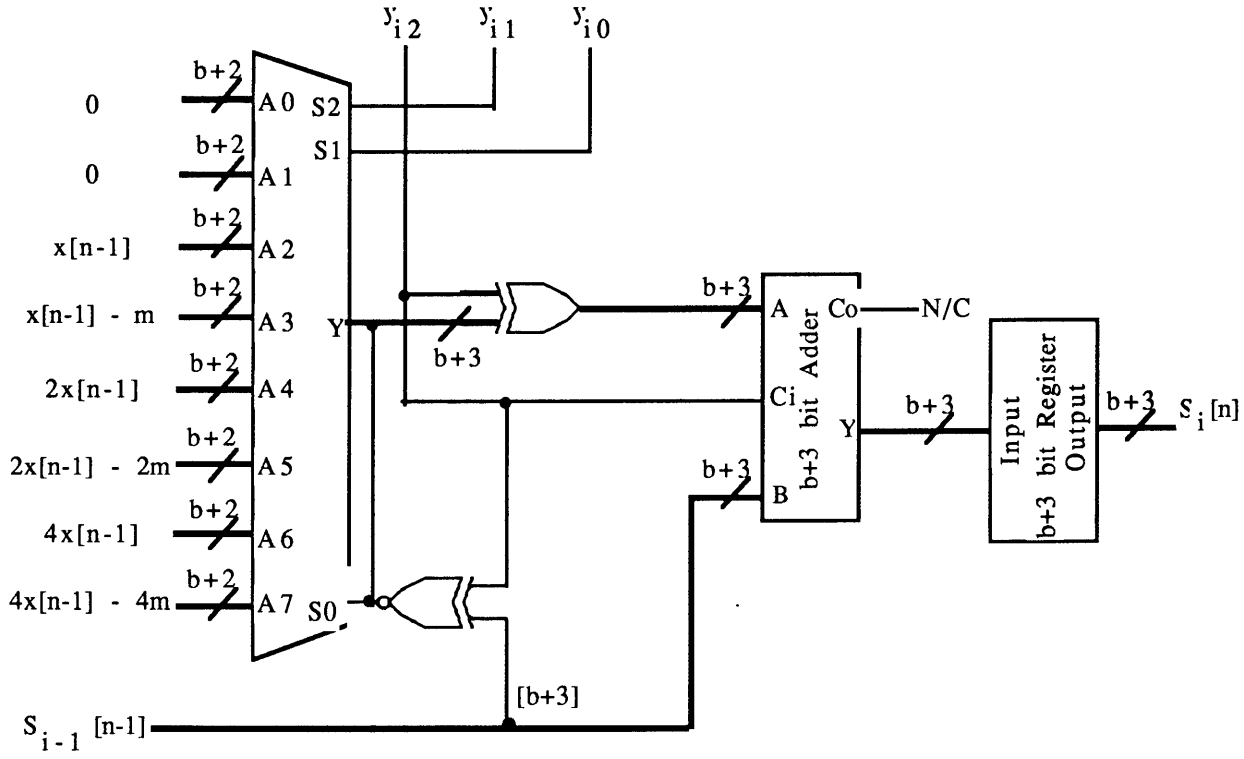


Figure 42 New Modified Septary Subtap

A good way to understand the modified digit set is to examine a number in the standard balanced septary representation and convert it to the modified digit set. A simple rule exists for the conversion: starting from right to left whenever a 3 occurs replace it with a -4 and carry 1, and whenever a -3 occurs replace it with a 4 and carry -1. The positive numbers of a two digit modified system are listed below

standard		modified		
b_1	b_0	a_1	a_0	decimal number
0	0	0	0	0
0	1	0	1	1
0	2	0	2	2
0	3	1	-4	3
1	-3	0	4	4
1	-2	1	-2	5
1	-1	1	-1	6
1	0	1	0	7
1	1	1	1	8
1	2	1	2	9
1	3	2	-4	10
2	-3	1	4	11
2	-2	2	-2	12
2	-1	2	-1	13
2	0	2	0	14
2	1	2	1	15
2	2	2	2	16
2	3	Not Possible		17

Examining the table of modified balanced radix 7 representations, the largest positive number that can be represented is $(22)_7$. In general, for any number of digits the largest positive number that can can be represented is $(2...2)_7$. Correspondingly, the largest magnitude negative number is that with all digits equal to -2. The total span of a J bit modified radix 7 number system is

$$\sum_{i=0}^{J-1} 2(7^i) + 1 + \sum_{i=0}^{J-1} 2(7^i) = \frac{2}{3}(7^J - 1) + 1$$

where the first term on the left accounts for the negative numbers, the second term for 0, and the final term for the positive numbers. Unfortunately, we lose approximately one-third of the span to make all of the digits powers of two. The resulting number of subtaps per tap is listed in the following table for several values of b.

b	Subtaps
2	1
3	2
4	2
5	2
6	3
7	3
8	4
9	4

The modified radix 7 representation requires fewer subtaps per tap than the balanced radix 5 representation for b equal 5 or 7.

Once again, the actual hardware design is very similar to the previous two designs. Using the formula at the end of section 4.1.3.1, b+3 bits are needed in the binary channels. The left shifts are performed by a hardwired 8-1 MUX although a custom left shifter[1] would be more efficient in hardware size and speed. One small advantage of using the 8-1 MUX is that the delay of an AND gate in the critical path is removed, although the added delay of an 8-1 MUX and 3 additional adder stages more than compensates in added delay. The final design is shown in figure 42. The coefficients are decoded as follows:

h_2	h_1	h_0	coefficient
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	4
1	0	1	-1
1	1	0	-2
1	1	1	-4

Any comparison of this design with other designs should consider that only two b bit busses are used. Limiting the number of global buses was the primary goal, and it has been achieved. The complete hardware summary is shown below

[1] In this case the rectangular grid would have transmission gates on three central diagonals.

Part Type	Number	Sizing	Transistors
1 bit Full Adder	b+3	9792b + 29376 μ^2	32b + 96
8-1 MUX	b+2	17952b + 84864 μ^2	50b + 220
1 bit Register	b+6	8160b + 48960 μ^2	24b + 144
XOR gate	b+3	4896b + 14688 μ^2	10b + 30
XNOR gate	1	4896 μ^2	8
Totals	---	40800b + 182784 μ^2	116b + 498

Global Bus	2b signal lines
Critical Path	XNOR + (4-1 MUX + XOR) + AND + b+2 bit adder + register
Throughput	$(1.9b + 16.78\ ns)^{-1}$

Architecture Summary for the balanced quinary subtap #2

Subtap Summary

The tables below list the size in μ^2, the number of transistors needed, and the latency for each of the three subtap designs using the new algorithm. All of numbers were obtained from a standard cell library. Figures 43 and 44 graphically summarize the size and transistor data.

		per subtap		entire tap	
b	J	size (μ^2)	transistors	size (μ^2)	transistors
2	2	107712	290	215424	580
3	2	135456	368	270912	736
4	3	163200	446	489600	1338
5	4	190944	524	763776	2096
6	4	218688	602	874752	2408
7	5	246432	680	1232160	3400
8	6	274176	758	1645056	4548
9	6	301920	836	1811520	5016

Balanced Ternary

b	J	per subtap size (μ^2)	per subtap transistors	entire tap size (μ^2)	entire tap transistors
2	1	151776	444	151776	444
3	2	182784	530	365568	1060
4	2	213792	616	427584	1232
5	3	244800	702	734400	2106
6	3	275808	788	827424	2364
7	4	306816	874	1227264	3496
8	4	337824	960	1351296	3840
9	4	368832	1046	1475328	4184

Balanced Quinary

b	J	per subtap size (μ^2)	per subtap transistors	entire tap size (μ^2)	entire tap transistors
2	1	264384	730	264384	730
3	2	305184	846	610368	1692
4	2	345984	962	691968	1924
5	2	386784	1078	773568	2156
6	3	427584	1194	1282752	3582
7	3	468384	1310	1405152	3930
8	4	509184	1426	2036736	5704
9	4	549984	1542	2199936	6168

Modified Balanced Septary

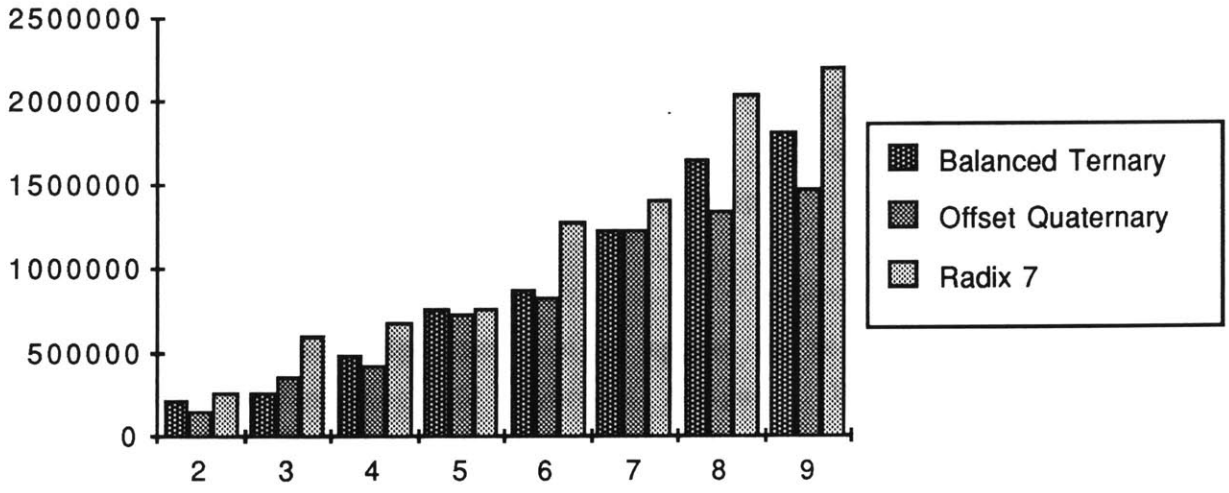


Figure 43 Size (μ^2) verses Number of Bits in the Binary Channels

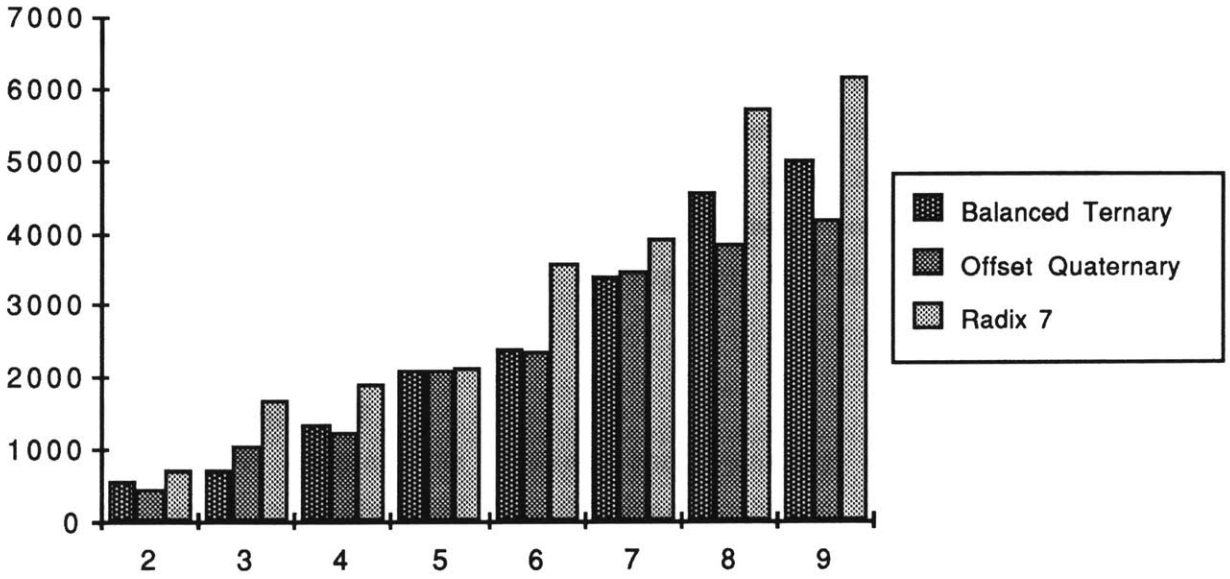


Figure 44 Transistors verses Number of Bits in the Binary Channels

b	Balanced Ternary	Balanced Quinary	Modified Septary
2	14.16 ns	17.86 ns	20.58 ns
3	16.06 ns	19.76 ns	22.48 ns
4	17.96 ns	21.66 ns	24.38 ns
5	19.86 ns	23.56 ns	26.28 ns
6	21.76 ns	25.46 ns	28.18 ns
7	23.66 ns	27.36 ns	30.08 ns
8	25.56 ns	29.26 ns	31.98 ns
9	27.46 ns	31.16 ns	33.88 ns

Latency through Subtap

In each case the new algorithm subtaps appear to be both larger and slower than the corresponding old algorithm designs. Unfortunately, the numbers can only be considered rough estimates of the actual hardware and speed of the new algorithm designs. Since all of the RNS designs discussed in this paper are intended for full custom implementation, the actual implementations would not be restricted to the parts in a standard cell library, and the subtap designs, both old and new algorithm, would be both smaller

and faster.[1] However, the new algorithm designs are even more disadvantaged by the standard cell library. Because there was no left shift block available, the left shifts were implemented by physically larger and slower multiplexors. The most discrepancy occurs for the modified radix 7 implementation where an 8-1 MUX was used instead of a 2-1 MUX, a left shifter, and an AND gate. If full custom designs were compared the new algorithm subtaps would be comparable or superior in both hardware and speed.

Putting it all Together

Earlier in this chapter the scaling and summing of each mini-convolution channel was discussed for the biased/unbiased algorithm. The same computation must be performed for the new algorithm also. Fortunately, the same scaling boxes can be used. The primary difference between the two cases is the form of the output at the final subtaps. For the biased/unbiased algorithm the output of a subtap is always normalized, but may or may not be biased. To compute an unbiased normalized version of the output requires only one clock cycle. Using the new algorithm, the output of a subtap is always unbiased, but may or may not be normalized. Depending on the range of unnormalized values that the output can span[2], several clock cycles are needed to generate a normalized version of the output.

Earlier, the output of a subtap was shown to vary within the range $\pm$ m* max(h_i). One of the fundamental assumptions of the new algorithm is that the members of the digit set are powers of two; therefore, h_i can be equivalently written as 2^k, and the output range written as $\pm$ 2^k m. Normalizing the output is performed by successively adding or subtracting decreasing powers of 2 times m. The top level block diagram of this algorithm is shown in figure 45.

---

[1] The area given for a standard cell part includes a boundary around the edges of the actual part to prevent violations of design rules. The area for a custom design including several parts will be significantly less than the sum of the areas of each part and their respective boundary layers.

[2] Determined by the maximum allowed digit in the coefficient decomposition

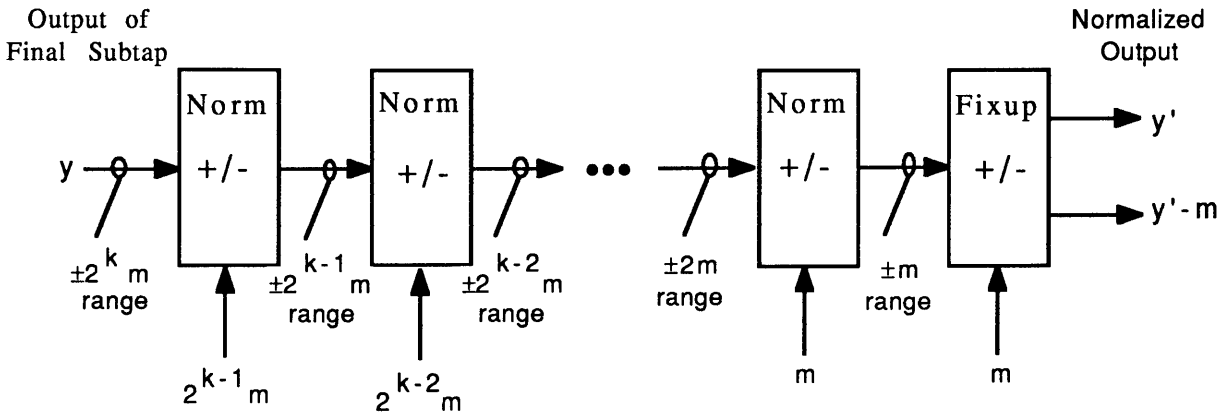


Figure 45 Normalizing Stage

A block diagram of a norm box is shown in figure 46. Each of the norm boxes operates in a manner similar to the subtaps of the filter. If the input to a norm box is positive, a negative multiple of the modulus is added to it; if the input is negative, a positive multiple of the modulus is added to it. At each step the range of the output is reduced by a power of two until the output falls into the range ± m. The fix block at the end of the chain operates in a similar manner, but adds the capability to output both unbiased and biased[1] versions of the output.

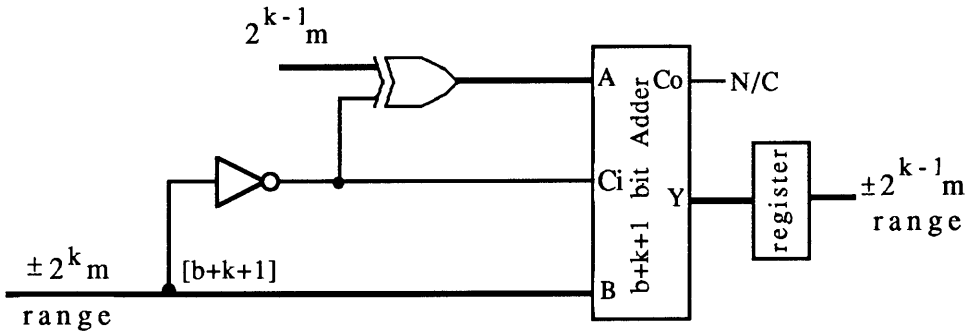


Figure 46 Norm Box

At the beginning of this section it was implied that the same scaling units designed for biased/unbiased algorithm can be used here also. This is true; however, it should be mentioned that scaling units can also be designed within the philosophy of the new algorithm. The new scaling units would add x, x-m and left shifted versions of each together to obtain an unnormalized

[1] Biased version of x is equal to x-m ignoring the b+1st bit.

scaled value that spans a certain range. A chain of norm boxes and a fix box would return the scaled value to the normalized range. There may slight hardware and aesthetic advantages to the new algorithm scaling units; however, they do exhibit an increased latency because of the added correction stages.

Binary to RNS Conversion Block

Now that architectures have been developed to compute a high-speed convolution sum within RNS, a method is needed to convert the data into a residue form at an equally high throughput and a low latency. Because a binary to RNS converter is needed for each modulus, some form of programmability is necessary to allow the same design to operate for any modulus. Programmability can be obtained with different levels of programming. The filter chain designs which are also needed for each modulus used the simplest form of programming, a single b bit number that could be loaded into a register or asserted to input pins. More complex programming would consist of loading several values into registers or blocks of memory. When comparing designs for the binary to RNS converter, the primary consideration becomes the level of programming. How much is it worth to eliminate tables?

Table Lookup Approach

The table lookup approach is useful because a binary to RNS conversion is a linear function, ignoring possible normalization. If the binary input to the filter is d bits and the residue channels are b bits, the residue value of the input is equal to the sum of the low order b bits of the binary input with the residue value of the high order (d-b) bits. The conversion of the high order bits can be performed by a table; the resulting conversion unit is shown in figure 47. Unfortunately, the 2^{d-b}xb bit table used could be very large and slow for large values of d. Because the large table is implementing a linear function, however, it could be replaced by a number of smaller tables. A similar approach for general linear functions of one variable has already been discussed in chapter 3.

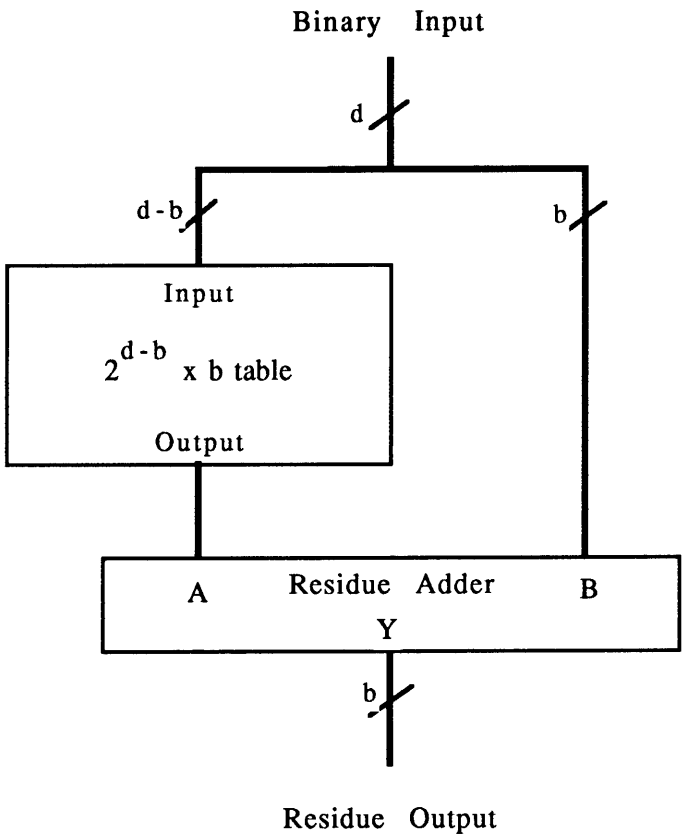


Figure 47 Table Lookup Approach for Binary to RNS Conversion

Using several smaller tables seems to be the best way to achieve a high throughput conversion, but some method is needed to efficiently modulo add the outputs of these tables. Since several versions of residue accumulators have been developed for the filter chains, these can be used as a starting point. In order to avoid drastic recoding of the high order d-b bits of the input, only the radix 2 and radix 4 accumulators[1] will be considered.

Because the radix 4 accumulator designs used an offset digit set {-2, -1, 0, 1}, the input must be recoded. The recoding algorithm is as follows: taking the high order (d-b) bits of the input in pairs obtain the standard radix four representation for the number, then starting at the low order digit if the digit is 2 replace it with -2 and carry 1 to the next place, if the digit is 3 replace it with -1 and carry 1, otherwise let the digit remain unchanged. An alternate

[1] A valid observation is that radix 2 and 4 accumulators were only developed for the biased/unbiased algorithm. Versions can also be developed using the new algorithm. A radix 2 accumulator can be implemented by removing the two's complement circuitry from the new radix 3 design. A radix 4 accumulator has a data path that is identical to the radix 5 design; the only difference is a more complex coefficient decoding.

method to recode the input is to add $(22...2)_4$ or $(1010...10)_2$. Although the result must be interpreted correctly, this method does generate the desired carry if a digit is 2 or 3. An example for the single radix four digit is shown below

00 (0)	01 (1)	10 (2)	11 (3)
<u>10</u>	<u>10</u>	<u>10</u>	<u>10</u>
10 (0')	11 (1')	1 00 (-2')	1 01 (-1')

Examining either conversion algorithm, the offset radix 4 form of the high order bits may contain an additional digit than the standard radix 4 representation.

Using the biased/unbiased algorithm, four values are needed at each accumulator $\{x, x+\mu, 2x, 2x+\mu\}$ Here x denotes the normalized mod m value of 2^i where i is the place of the low order bit of the bit pair in the total input. These values must be loaded into addressable memory or, more practically, into a tapped shift register. For a d bit binary input and b bit residue channel, the total number of b bit stored values is $4\lceil (d-b)/2 \rceil$.

If the new algorithm is used instead, only x and x-m are needed at each accumulator. These values would also be loaded into addressable memory or a tapped shift register. For a d bit input and b bit residue channel, the total number of b bit stored values is only $2\lceil (d-b)/2 \rceil$. Unfortunately, every bonus has an equal and opposite penalty. Although only one-half as many registers are needed, the new algorithm uses an extra correction stage which not only adds hardware, but also increases the latency of the conversion.

Both of the offset radix 4 design require a (d-b) bit binary adder to perform the conversion between the standard digit set and the offset digit set. The adder increases both the hardware size and latency of the conversion. If d-b is on the order of b, the add can probably occur within a single clock cycle; otherwise, the binary adder must be pipelined increasing the latency even more.

The digit set conversion can be avoided entirely if the standard radix 4 digit set {0, 1, 2, 3} is used. Although either radix 4 accumulator can be modified to use the standard digit set, both designs will require more stored values and contain larger selectors. The decrease in latency is paid for in hardware size.

No Table Approach

If simple programmability is more important than both hardware size and latency, binary to RNS conversion units can be developed using the simple programmable residue doubler design in an extended residue multiplier. Because the b bit residue value of 2^b is known[1], it can be multiplied by the d-b high order bits of the input using the residue multiplier design from chapter 3; the b low order bits of the input can be used as an

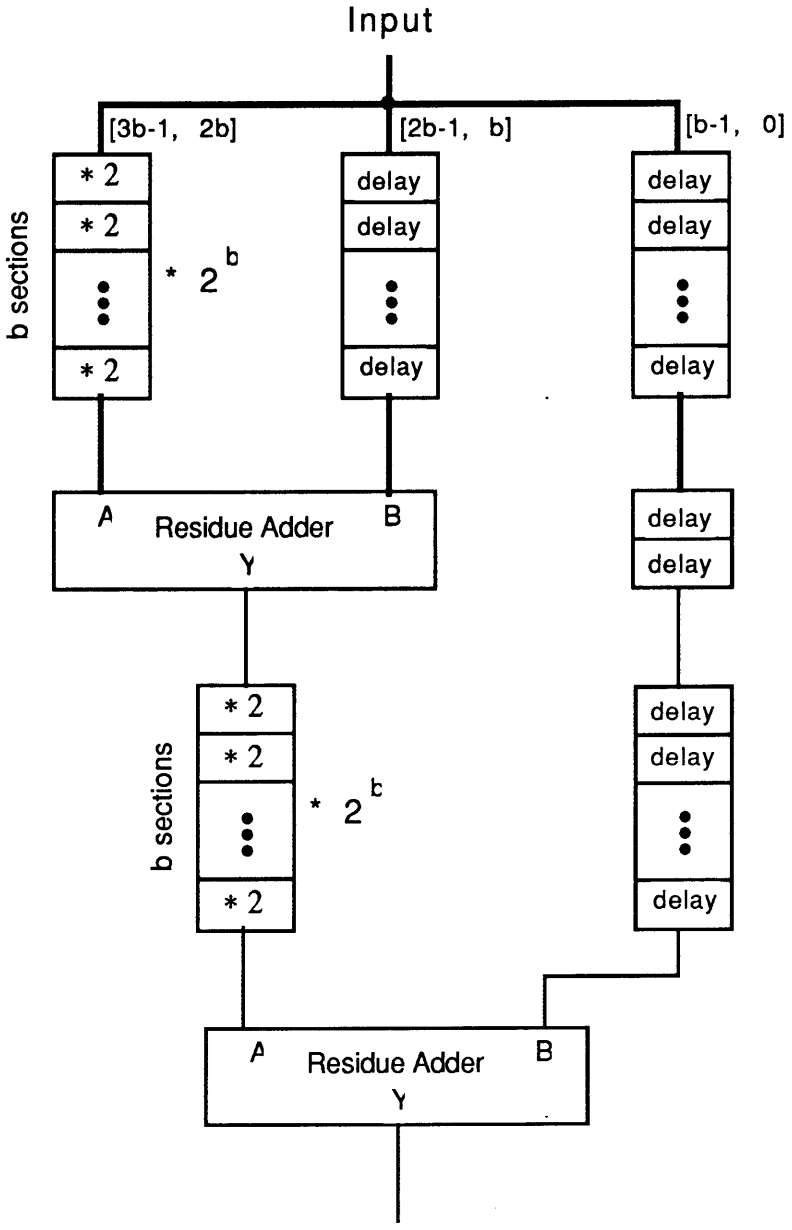


Figure 48 No Table Binary to RNS Conversion

[1] $2^b \equiv \mu \bmod m$, think about it... Also, if the moduli set is chosen so that all moduli are greater than 2^{b-1}, the biased form of 2^b can be formed by left shifting μ.

unbiased seed for the first accumulator in the multiplier. The result is a simple programmable binary to RNS converter; only μ needs to be loaded or asserted to an input.

Another simple programmable conversion unit using residue *2 blocks can be designed by segmenting the binary input, starting at bit 0, into b bit sections. Using *2 blocks the higher order segments can be multiplied by the appropriate powers of two and the result summed. A block diagram of the design for $\lceil d/b \rceil = 3$ is shown in figure 48. This design would be competitive primarily for small $\lceil d/b \rceil$.

Residue to Binary Conversion

An RNS to binary conversion unit is needed to put the results from all of the residue channels back together to form a binary number. Unfortunately, the RNS to binary conversion is significantly more complex than the binary to RNS case. Because the RNS digits are uncoupled and unordered[1], there is no simple conversion algorithm that would allow each digit to be converted independently. Two algorithms that have been extensively discussed in the literature are the Chinese Remainder Theorem and the Mixed Radix conversion algorithm.

Chinese Remainder Theorem

In chapter 3 the Chinese Remainder Theorem was presented to show the mathematical link between the residue representation and a conventional weighted number representation. This equivalence is as follows:

$$x = \left(M_1 \langle M_1^{-1} \rangle_{m_1} x_1 + M_2 \langle M_2^{-1} \rangle_{m_2} x_2 + \ldots + M_r \langle M_r^{-1} \rangle_{m_r} x_r \right) \bmod M$$

where $M = \prod m_i$, $M_i = M/m_i$, and $x_i = x \bmod m_i$. The computations required to evaluate the CRT expression are multiply-accumulates modulo M; however, M is on the order of 2^{rb} which could be very large. To avoid modulo M arithmetic other algorithms must be investigated.

---

[1] If each residue in an RNS system is considered to be a digit, the digits are uncoupled because there are no carries. This feature allows us to perform smaller computations on each digit without any links between the digits. A more familiar weighted number system such as the decimal number system has coupled digits.

Mixed Radix Conversion

A standard way to avoid the modulo M arithmetic in the CRT is to use the mixed radix conversion algorithm. With this algorithm the residues are first converted to an intermediate mixed radix[1] representation that is then evaluated to find a standard fixed radix value. The calculations required to convert from residue to mixed radix are all b bit modulo m_i and the remaining calculations are conventional binary.

The Algorithm

To simplify the conversion process the radices in the mixed radix representation are chosen to be equal to the moduli in the moduli set. The best way to motivate the algorithm is to reverse engineer it. First, the following notation must be defined

Γ_i = ith mixed radix coefficient

m_i = ith modulus

x_i = ith residue coefficient

γ_{ij} = $\langle \Gamma_i \rangle_{mj}$

m_{ij} = $\langle m_i \rangle_{mj}$

Assume x is already in the mixed radix form

$$x = \Gamma_0 + \Gamma_1 m_1 + \Gamma_2 m_1 m_2 + \Gamma_3 m_1 m_2 m_3 + \ldots$$

If there are r moduli in the moduli set, there will be r digits in the associated mixed radix representation. Taking residues of both sides of this equation for each modulus in the moduli set, the left side (x mod m_i) equals x_i, and the right side is some function of the Γ_i's. With the x_i's known we can solve for the Γ_i using residue arithmetic. The first three digits are shown below

$x_1 = \Gamma_0$ $\qquad$ $\Gamma_0 = x_1$

$x_2 = \langle \Gamma_0 + \Gamma_1 m_1 \rangle_{m2}$ $\qquad$ $\Gamma_1 = \langle x_2 - \Gamma_0 \rangle_{m2} \langle m_1^{-1} \rangle_{m2} = [(x_2 - \gamma_{12}) * m_{12}] \bmod m_2$

$x_3 = \langle \Gamma_0 + \Gamma_1 m_1 + \Gamma_2 m_1 m_2 \rangle_{m3}$ $\qquad$ $\Gamma_2 = [((x_3 - \gamma_{13}) * m_{13} - \gamma_{23}) * m_{23}\ ] \bmod m_3$

---

[1] Previously, for coefficient decomposition fixed radix number systems were defined. Mixed radix number systems are similar except that the radix α can vary from place to place. If x_i are the digits of a mixed radix number with radices α_i, the value of the number is computed as follows:

$$x = \sum_{i=0}^{J-1} \left(\prod_{j=0}^{i} \alpha_i \right) x_i$$

Much of the computation for the residue to mixed radix conversion can be performed in parallel. The example below shows the conversion process for a simple 3 moduli RNS.

Example	$m_1 = 3$	$m_2 = 4$	$m_3 = 5$
	x1 = 1	x2 = 0	x3 = 4
$\Gamma_1 = 1$		$-(\gamma_{12} = 1)$	$-(\gamma_{13} = 1)$
		3	3
		$*(m_{12}^{-1}=3)$	$*(m_{13}^{-1}=2)$
		1	1
$\Gamma_2 = 1$			$-(\gamma_{23} = 1)$
			0
			$*(m_{23}^{-1}=4)$
			0
$\Gamma_3 = 0$			

A top level block diagram of the required computation is shown in figure 49.

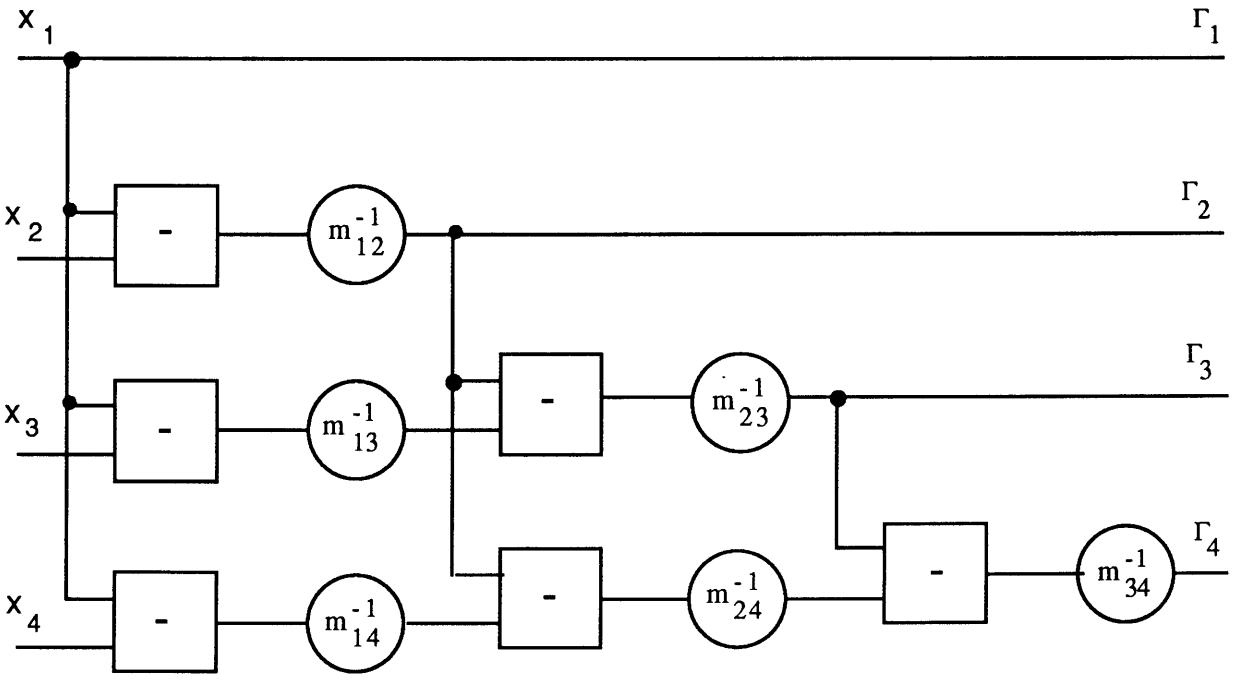


Figure 49 Mixed Radix Conversion Algorithm (for 4 residue channels)

The Hardware

The primary consideration for the hardware design is the level of programmability. Because the conversion process includes each modulus in the moduli set, the conversion hardware sets a limit on the number of allowable moduli. The addition of another modulus requires a new design. Although it would be nice for testing if the moduli set could be arbitrarily programmed, the feature is not really required. If programmability is omitted, the optimum moduli set[1] can be hardwired into the design. The advantage of hardwiring the moduli set is that the residue multiplies can be performed by table lookups without having to load the tables.[2] To simplify the following discussion the original biased/unbiased algorithm will be used for residue arithmetic. The new unbiased range algorithm could also be applied, but it would only complicate the presentation.

Except for timing registers, the data flow graph in figure 49 is a top level block diagram for the mixed radix conversion hardware. Two designs are needed, one a residue subtractor and the other a residue scaler.

The residue subtractor is a very simple design. If both inputs are available in unbiased form, the subtraction can be performed by two's complementing the subtrahend and adding the complemented form to the minuend (figure 50). Because the inversion will generate the biased form of the negative residue, the biased/unbiased addition requirement is satisfied, and the carry out of the adder will indicate the state of the output. The only problem with this subtractor is that an overflow will cause undefined results. To guarantee that no overflow occurs, the moduli set must be ordered (ie $m_1 < m_2 < m_3 < ... < m_r$).

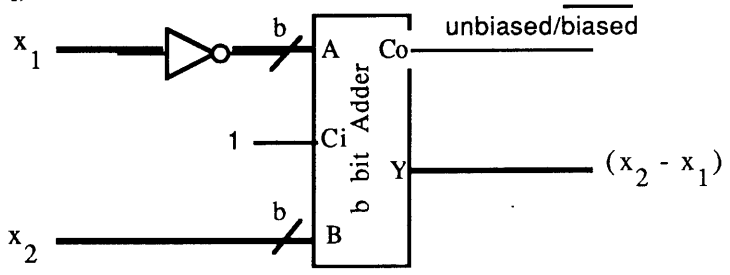


Figure 50 Residue Subtractor (Unbiased/Biased Algorithm)

---

[1] The optimum moduli set is that set of reletively prime moduli representable in a b bit binary channel that have the maximum product. See chapter 5

[2] The number of multiplies that would occur in a practically sized MRC would require a large number of memory in which to store tables.

The residue scaling unit is also a simple design. Because scaling is a linear operation, the design is very similar to the conversion unit for the high order bits of the binary to RNS conversion. The standard radix 4 accumulator described earlier can be used in a configuration similar to that in figure 24.

Putting the two units together, the mixed radix conversion can be completed with a throughput equal to the filter chains. To place the input into its final binary form, several binary multiplies must be performed to multiply out the mixed radix coefficients. These multiplies can be performed by standard binary shift and add multipliers with the individual adders pipelined to match the throughput of the filter chains. Because the mixed radix digits are weighted[1], the low order digits may be neglected.

[1] Assuming that all moduli are on the order of 2^b, the most significant mixed radix digit multiplies $\sim 2^{rb}$; the least significant multiplies 1. If several moduli are used, the low order digits will be noise.

Chapter 5 Design Aid

Now that some basic architectures have been designed and analyzed, a tool can be developed to present the hardware size and speed data in more friendly form. Given the number of bits in the input and coefficients, and the length of the filter, the program will output the optimal architectures for both old and new algorithms. Originally, the intention was to have the user input a size/latency cost function. After some consideration, I decided to output all possible optimal size/speed designs, and allow the user to weight the alternatives. In addition, no distinction will be made between the two types of designs because it was not possible to accurately estimate the hardware or speed of these designs or characterize the advantage of having fewer buses.

The algorithm consists of two fairly distinct sections: an algorithm to compute the minimum number of moduli channels for each bitwidth and an algorithm to prune the design space.

Moduli Selection Algorithm (Basic RNS)

The number of bits in the coefficients and input and the length of the filter determine the required dynamic range for the filter. If the input is d bits, the coefficients e bits, and the length N, the total number of bits of dynamic range is $(d + e + \log_2 N)$.[1] Starting with this dynamic range requirement, an optimum moduli set[2] can be chosen for each b within the range [3 , 9]. The optimum moduli set itself is not so important; however, the number of moduli, r, in the sets for different values of b determines the number of residue channels that are needed. However, it is much simpler to solve the reversed problem: given b and r find the optimum moduli set and its product. The moduli selection algorithm can then be used to solve for r given b and a target moduli product.

A first attempt at optimal moduli selection uses an exhaustive search[3] of all sets of r relatively prime numbers requiring b or fewer bits. The search proved to be extensive for large b and r and required a significant amount of

---

[1] The $\log_2 N$ term can be significant for a long FIR filter; it is the price of exact calculation (no rounding).

[2] Optimum moduli set implies the set of relatively prime moduli representable in b bits that has the highest product of any set of relatively prime moduli representable in b bits with the same number of moduli in the set.

[3] See Appendix for code

computer time. This prompted a search for a more efficient algorithm that used some of the properties of an optimal moduli set.

To improve the speed of the algorithm, it is worthwhile to consider some of the properties of the optimum moduli set and limit the search a bit. A simple way to motivate a better algorithm is the following: for a certain modulus m_i to be included in the moduli set, the advantages of its being included must be greater than the disadvantages of other potential moduli being excluded because they are not relatively prime to m_i. A first observation is that 2^b should always be included in any optimum moduli set. Because 2^b contains only factors of 2 , it can only exclude other even numbers. Since 2^b is the highest even number in our potential moduli set and only one even number can be included in the final moduli set, 2^b should be in this set. A second observation is that the remainder of the moduli set will contain odd numbers that are as large as possible. This information can be added to the exhaustive search algorithm to substantially reduce the search time; however, the search still becomes unwieldy for larger b and r.

To improve on the reduced exhaustive search algorithm it is possible to directly choose the moduli set and avoid a lengthy search. As a first attempt at direct selection, start with $2^b - 1$ as the first odd in the set and step down the remaining odd numbers adding those that are relatively prime to the set until the requisite number r moduli have been chosen. The inspiration for this algorithm is similar to that above which included 2^b because it is the highest number with a factor of 2. Only the highest number having a factor of 3 will be included along with the highest having a factor of each prime 5, 7, 11, etc. Unfortunately, this algorithm does not always give the best answer. It works for b = 2, 3, 4, 5, and 7, but for some cases of b=6 or 8 it gives a suboptimal result.

A deeper investigation of optimum moduli sets shows that it is occasionally optimal to exclude a larger odd number in order to include two smaller ones. For example, if the highest number having a factor of 3 also contains a factor of 5 (which is the case for b = 8) then this number not only excludes all smaller factors of 3 but also all smaller factors of 5. The problem can be more easily visualized if a moduli set with r moduli is thought of as been generated by adding a modulus to the moduli set of r-1 moduli. At some point the product of the moduli will be increased more by excluding the highest factor of 3 and 5 and including the second highest factor of 3 and the

second highest factor of 5 rather than including the next smaller odd number that is relatively prime to the existing set. A numerical example is shown below

8 Bit Moduli --- 6 Moduli in Set

256, 255, 253, 251, 247, 241

8 Bit Moduli --- 7 Moduli in Set

256, 253, 251, 249, 247, 245, 241

In this example 255 contains both factors of 3 and 5. When the seventh modulus is added it becomes advantageous to omit 255 from the moduli set and add the next highest factor of three, 249, and the next highest factor of five, 245. This is because 255*239 = 60945 and 249*245 = 61005. The replacement could have been anticipated by realizing that 255 is a double factor odd[1] and calculating LOW which equals the product of the second highest odds containing the factors in the double factor odd divided by the double factor odd. In this case LOW= 249*245/255 = 239.2. If the next number that is relatively prime to the existing set (239 in this case) is less than LOW, then the double factor prime should be omitted instead.

A heuristic algorithm was developed to avoid double factor primes without explicit factoring. First, a moduli set consisting of 2^b and the largest r-1 relatively prime odds less than 2^b is generated. Then, each odd in the set is sequentially excluded, and the largest r-1 relatively prime odds are chosen (omitting the excluded one from the search space). The product of the new moduli set is calculated, and if greater than the previous product, this set becomes the current one from which odds are sequentially excluded. When all odds have been excluded from a current set without any of the resulting moduli sets having a greater product, the search ends with the current set as a result.

The excluding heuristic algorithm gives the correct moduli set for practical values r and b. For some very large values of r, the exclusion must be ordered in ordered for the algorithm to converge on the optimal set. The

---

[1] More properly, 255 contains two small factors (3 and 5)

first error occurs for b = 8 and r = 38. The error can be ignored for all practical uses.[1]

Now that a method to find the optimum moduli set given r and b has been developed, we need a way to find the sufficient number of b bit moduli needed to achieve a given dynamic range. Since all moduli within a b bit channel will be less than or equal to 2^b, an initial lower guess is r = (# bits of range needed / b). Starting with this value of r, the heuristic moduli selection algorithm can be called with successively higher values of r until a moduli set is found that has the sufficient range.

The Design Aid

Given the number of moduli needed for different bitwidth moduli and the size/latency data for the different bitwidth residue subtap designs, a set of (size, delay) pairs can be formed that can be searched to find potential optimum designs. First, the size data listed in chapter 4 is multiplied by the appropriate number of moduli required for the respective bitwidths to obtain a size number for the entire filter tap. Because all of the subtaps within a tap all operate in parallel, the filter latency is equal to the subtap latency for any number of moduli channels. Second, the scaled size and latency figures are grouped into pairs that make up the possible design space. Four design types were investigated using the biased/unbiased algorithm: binary, balanced ternary, offset quaternary, and balanced quinary. Three design types were investigated for the using the new algorithm: balanced ternary, balanced quinary, and modified balanced septary. For each design type, seven implementations are considered, one for each moduli width from b=3 to b=9. So, the design space of the old algorithm contains up to 28 designs, and the design space of the new algorithm contains up to 21 designs[2].

The search algorithm used to prune the space of designs operates on a principle of finding dominant designs that exclude others. When two designs are compared, if one has both a smaller size and a lower latency than another, the first is said to dominate. The dominant design would always be chosen, and the other can be pruned. If one has a lower delay and the other has a

---

[1] The dynamic range obtained from 37 8 bit moduli is 274.65 bits of precision. It is unlikely that any application would need this precision.

[2] In general, some of the lower values of b will be excluded because there are not enough moduli representable in b' bits to acheive the desired dynamic range.

smaller size, the two are said to coexist, neither excludes the other. If each design in the design space is compared to the others, a group of coexisting dominant designs will remain. These designs will have the optimum size/speed combinations. A good way to visualize the process is to imagine a 2-D scatter plot of the design space with size on one axis and delay on the other. We only want to keep those designs that plot closest to the origin.

Discussion

Running the algorithm gives the results that would be anticipated from looking at the charts in chapter 4. For the biased/unbiased algorithm the offset quaternary design usually has the smallest size and the largest delay of the optimum set, and the size charts show the offset quaternary design to have the smallest size for b=4 to 8. The other designs in the old algorithm optimum set are usually the balanced ternary and the binary. Because the delay increases with the radix, this is also expected.

One design type in each algorithm is always dominated by others. For the old algorithm, the balanced radix 5 design is dominated by the offset radix 4 design for b less than 9. Although the subtap delays are the same for all values of b, the advantage of the radix 5 design is not realized until 9 bit moduli are used. For the new algorithm, the modified balanced septary design is dominated by the balanced quinary design for all values of b.

Chapter 6 Standard Binary Arithmetic with Pipelining

Now that the RNS designs have been presented and analyzed, another alternative will be presented using standard binary arithmetic. Using some of the techniques developed for residue filter taps and extensive pipelining, a conventional design can be implemented that operates at a higher throughput than the residue designs. Although no general decisions will be made between the residue and standard implementations, this design would be a good starting point for any comparisons.

Development of architecture

The architecture design is based on two concepts. The first is coefficient decomposition to avoid multiplies, and the second is pipeline registers inserted in the carry chain of long binary adders to increase throughput. The coefficient decomposition concept has been thoroughly discussed within the residue filter discussion. The primary difference here is that overflow cannot be permitted within standard binary arithmetic. A sufficient number of bits must be included in the binary channels to prevent overflow. The pipelined adder is the real innovation of the binary design. The adders can be pipelined to a granularity of single bit adders for a throughput of (1 adder delay + 1 register delay)$^{-1}$. For the residue problem the adders can not be pipelined because the high order carry out is needed to condition the next stage for the old algorithm designs and the high order bit is needed to condition the next stage in the new algorithm designs. For the residue designs, all of the calculation has to be performed within a single clock cycle.

Filter Tap

Using the transpose filter form, the temporary results move along a data path consisting of N adder/register pairs. A single adder/register combination is shown in figure 51. By shifting some of the registers to before the adder, the adder becomes pipelined as in figure 52. Performing the same register shift to all adder/register combinations in the data path preserves the overall data flow.

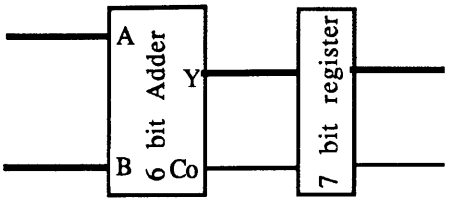


Figure 51 Throughput Limiting Data Path in a Transpose Form FIR Filter

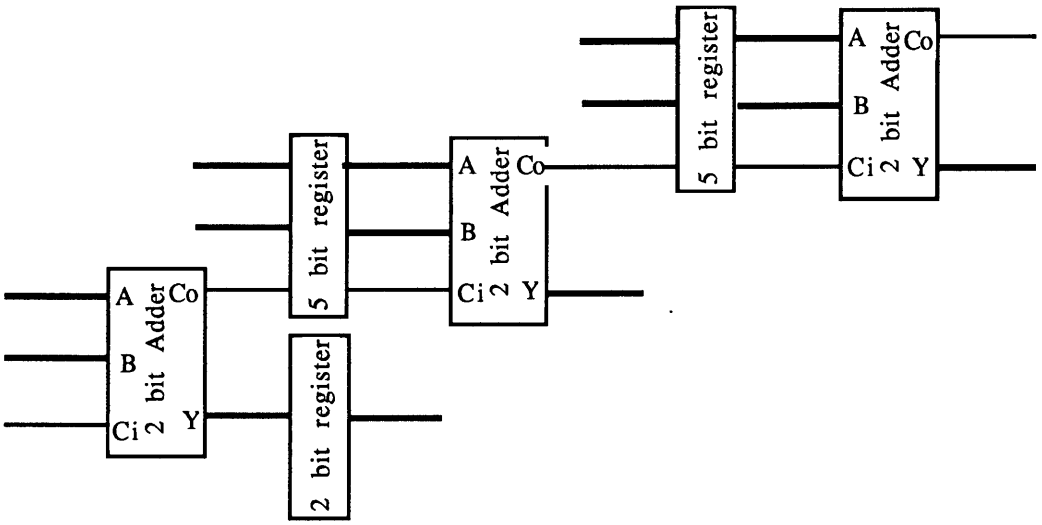


Figure 52 Increased Throughput Binary Adder/Register Combination

In the figures, a six bit adder is pipelined into three two bit adder sections. A first observation is that the number of registers increases when they are shifted to the inputs of the adder. This should be expected because there are two inputs for each output, and extra registers are needed for the carries. In general, a b bit adder/register pipelined into d bit sections, where d|b,[1] requires b one bit adders and 2b + (b/d) - d - 1 one bit registers.[2] Increases in throughput are bought with increases in hardware. Another observation is that the adder is operating on three different adds at the same time. Assuming the data is being clocked through, the first section is operating on the current inputs, the second section on the one previous inputs, and the final section on the two delayed inputs. Something is needed to guarantee that the inputs to the adder are staggered in time.

---

[1] d|b read d divides b (integer divide without remainder)

[2] For d=1, the number of registers = 3b - 2; for d = b/2, the number of registers = 3b/2 + 1.

Binary Subtap

To actually design a filter tap, a coefficient decomposition must be chosen. The hardware is simplest if the binary decomposition is used. Each subtap consists of a b bit adder and c AND gates as shown in figure 53. An input stage of registers stagger the input is shown in figure 54. With (b/d) sections per adder, $((b/d)^2 + (b/d))/2$ registers are needed for the input stage. Because all taps receive the staggered input, the adders will always receive the proper inputs.

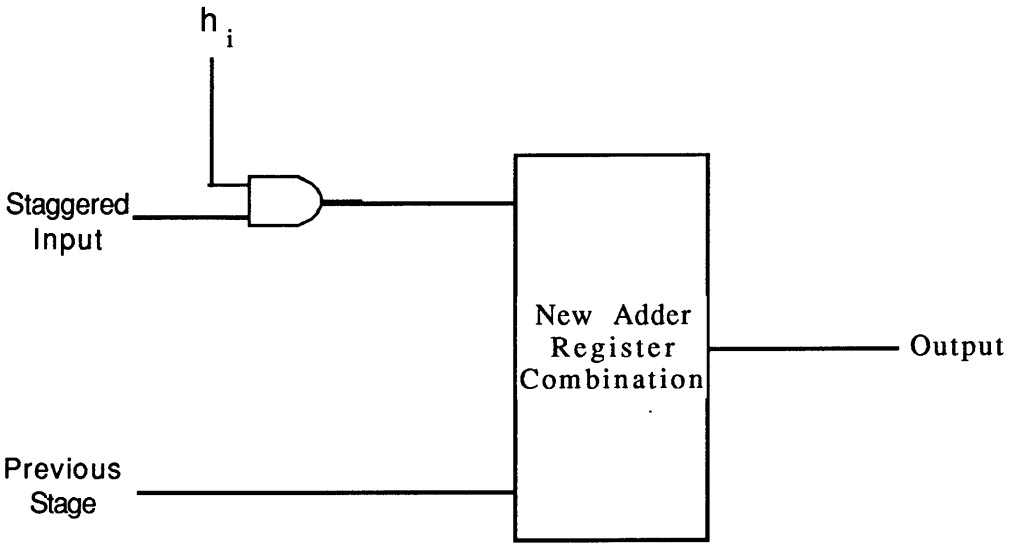


Figure 53 Binary Subtap using Pipelined Binary Arithmetic

Balanced Ternary

The advantage of the higher radix is that fewer subtaps are needed per tap. The balanced ternary design uses the same method used by the residue designs to two's complement the input. The design needs an additional c XOR gates to invert the input; the same signal gating the XOR gates is fed into the carry in of the adder/register combination.

Higher Radices

It is possible to go to even higher radix decompositions of the coefficients than balanced ternary; however, the designs complicate slightly. Although the inputs is broadcast to the taps in different time slices, the coefficient remains the same for all time slices, and scaling all time slices by factors of

two can be performed by shifting the input. A general form for the higher radix designs adds a left shifter to the balanced ternary design. Using this form and the appropriate coefficient decoding, offset quaternary, balanced quinary, and modified balanced septary subtap designs can be implemented.[1] Because there is no feedforward from the previous subtap, the throughput for the subtap is limited only by the d bit adder section and a register. The additional hardware does not decrease the throughput of the subtap. There is some difficulty, however, in synchronizing staggered versions of the input that have been scaled by the digits of the decomposition radix.

Shift Add Reconstruction

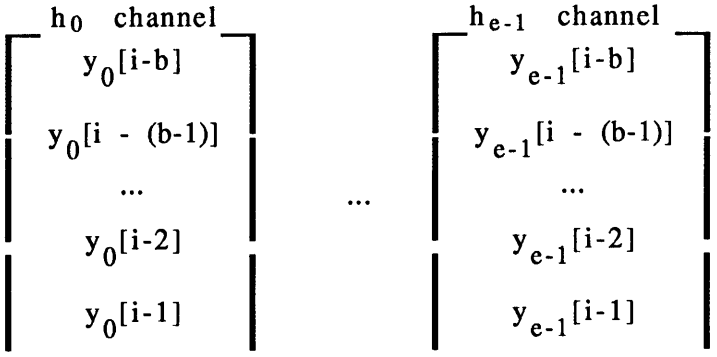
With filter taps constructed out of the parallel subtaps described above, the output of the final stage will not be in a "friendly" form. Each of the J mini-convolution chains will output a b bit result that consists of b/d d bit slices of different time results. There is the temptation to place an output stage[2] after each final subtap to put the result of each mini-convolution chain back together in time. When the scaling and summing of the mini-convolution chains at the same throughput as the filter is considered, the output stage does not seem as appealing. To use pipelined adders, the inputs must be staggered in time; however, this is the form in which they are output.

The following discussion focuses on the simplest case of reconstruction, binary coefficient decomposition with single bit adder sections. In this case the number of mini-convolution chains, J, is equal to the width of the coefficient, e, and the output of each mini-convolution chain consists of b 1 bit time slices. Assuming that there are J mini-convolution chains, each with a b bit output, the data arrives at the outputs as b/d d bit sections of time delayed data. The 0th and e-1st outputs are shown below

---

[1] Because no left shift block is available in the standard cell library, an explicit design is omitted. For all of these designs number of subtaps needed for e bit coefficients can be found in the tables of chapter 4.

[2] The output stage would consist of register chains similar to the input stage.



Focusing on the i-1st time slice, one bit can be grabbed from each channel's output. If the these bits are grouped in descending order $\{y_{e-1}[i-1], y_{e-2}[i-1], ..., y_1[i-1], y_0[i-1]\}$, the first slice into the adder is obtained. Because the output of the h_j channel is multiplied by 2^j, the group of bits is properly ordered in an e bit binary word. At the next output time, the i-2nd group contains results from this same time slice. The e bit binary word obtained from this output must be left shifted one place and added to the previously obtained e bit binary word. At the next output time the i-3rd group contains results from this same time slice. The e bit word obtained must be left shifted two places and added to the previous partial sum. This process continues until the final group that contains results from the same time slice arrives at the high order bits. All total, b shifted e bit numbers are added together to generate a single binary result. The hardware required to perform the reconstruction is b e bit adders; the latency of this hardware is b+e clock cycles.

A similar algorithm can be used to reconstruct for the case: offset quaternary coefficient decomposition with 2 bit adder sections. Using any other combinations significantly complicates the reconstruction process, especially any scaling by numbers other than powers of two. However, using pipelined adders the results can be reconstructed with the same throughput as the filter chain.

Hardware Required/Contrast with RNS architecture

Similar to the comparison between different residue designs, the hardware comparison that follows will focus primarily on the filter taps because of the assumed large number of taps. In any case, the input stage and reconstruction hardware will be smaller than their counterparts, the binary to RNS converter and the RNS to binary converter. One difference is that the size of the RNS hardware was determined by the total dynamic range; the size

of the massively pipelined binary design is a function of both the coefficient width and the combination of the input width and the filter length.

For any exact[1] FIR filter the total required dynamic range is d + e + $\log_2$ N where d = input width, e = coefficient width, and N = length of the filter. The residue FIR filter uses r b bit moduli channels with r and b chosen such that the product rb is just greater than d + e + $\log_2$ N. With the offset quaternary implementation, each residue filter tap uses a (b/2)b wide data paths. The complete filter is r(b/2)b bits wide. The pipelined binary filter (binary implementation) uses e channels each (d + $\log_2$N) bits wide. The complete pipelined binary filter (binary implementation) uses (d + $\log_2$N)e bits wide.

For a rough comparison, assume d = 8, e = 8, and N = 10000. The optimum size RNS design has 5 6 bit moduli with the offset quaternary implementation. The total width of the RNS data path is r(b/2)b = 5(6/2)6 = 90 bits wide. The offset quaternary binary filter would have 4 23 bit channels[2] for a total width of 92 bits. Both designs would involve similar shift and add stages, but the RNS design would require 2 or 4 buses to broadcast versions of the input to the subtaps. In addition, the RNS design would require large conversion stages at the beginning and end of the filter.

General Discussion

More detailed comparisons would be needed to actually make any decisions between the RNS and the pipelined binary designs. Based on the limited comparison that was performed, the pipelined binary designs seem very competitive to the comparable[3] residue implementation from a hardware sizing standpoint. Also, the throughput is significantly higher than the RNS designs. Whether further investigation shows the pipelined binary design to be superior or inferior, it would provide a yardstick with which to measure the RNS designs.

---

[1] Exact implies that no rounding occurs.

[2] The coefficients are decomposed into e/2 quaternary digits, and width of the channel data paths must be increased by one because the maximum coefficient now has a magnitude of 2.

[3] using the same decomposition radix

Chapter 7 Conclusion

With a cursory introduction to RNS, it is easy to become excited about the great potential for high speed computation of any signal processing algorithm requiring only addition and multiplication. After a deeper study of the topic, it rings clear, however, that RNS is only useful for a very limited number of applications. In part, the limitation stems from the problems with scaling and magnitude comparison, but the largest constraint on the general use of RNS is the huge overhead of the conversion units. Even for applications for which RNS is ideally suited such as the FIR filter problem, the size of the problem must be sufficiently large to warrant applying RNS techniques.

If more efficient conversion units can be designed, maybe the space of RNS applications could increase. The designs in chapter 4 are in the proper direction, away from the infamous table lookup solution to any RNS problem. However, a substantial number of computations are required for either the CRT or the MRC, and, regardless of the efficiency of the computational units, there is a large amount of computation to be performed.

The massively pipelined binary concept may prove to be superior to RNS techniques even on the problems that RNS is good at. The massively pipelined designs are faster in all cases, and the initial hardware estimates seem comparable. Obviously, more research should be done in this area before any global decisions are made concerning the two methods.

Assuming RNS does have merit, there is much more work to be done to extend the little that I have done. If custom VLSI versions of the designs are compared, I believe that the new algorithm designs would be better than the biased/unbiased designs in hardware size and possibly also speed. More research should be done on the new algorithm designs in general.

Throughout all of the architecture discussions, two's complement form was assumed for all signed number representations. As a result, the arithmetic units were all designed to operate on two's complement numbers. But, what if balanced ternary, offset quaternary, or balanced quinary representation is used for all numbers in a design; could more efficient arithmetic units be developed?

Another idea is to use redundant number representations. Although more logic is required for the arithmetic units, the shorter propogation delay

could be used to increase the throughput of the RNS filters. Maybe this technique could give RNS the needed edge in the battle with a pipelined conventional design.

Appendix 1 Dynamic Range for Optimum Moduli Sets

Bit Length	Moduli	Product	Bits of Precision	Bits Used	Efficiency	Increase
3	1	8.00000000E+00	3.0000	3	1.0000	
	2	5.60000000E+01	5.8074	6	0.9679	7.00
	3	2.80000000E+02	8.1293	9	0.9033	5.00
	4	8.40000000E+02	9.7142	12	0.8095	3.00
4	1	1.60000000E+01	4.0000	4	1.0000	
	2	2.40000000E+02	7.9069	8	0.9884	15.00
	3	3.12000000E+03	11.6073	12	0.9673	13.00
	4	3.43200000E+04	15.0668	16	0.9417	11.00
	5	2.40240000E+05	17.8741	20	0.8937	7.00
	6	7.20720000E+05	19.4591	24	0.8108	3.00
5	1	3.20000000E+01	5.0000	5	1.0000	
	2	9.92000000E+02	9.9542	10	0.9954	31.00
	3	2.87680000E+04	14.8122	15	0.9875	29.00
	4	7.76736000E+05	19.5671	20	0.9784	27.00
	5	1.94184000E+07	24.2109	25	0.9684	25.00
	6	4.46623200E+08	28.7345	30	0.9578	23.00
	7	8.48584080E+09	32.9824	35	0.9424	19.00
	8	1.44259294E+11	37.0699	40	0.9267	17.00
	9	1.87537082E+12	40.7703	45	0.9060	13.00
	10	2.06290790E+13	44.2297	50	0.8846	11.00
	11	1.44403553E+14	47.0371	55	0.8552	7.00
6	1	6.40000000E+01	6.0000	6	1.0000	
	2	4.03200000E+03	11.9773	12	0.9981	63.00
	3	2.45952000E+05	17.9080	18	0.9949	61.00
	4	1.45111680E+07	23.7907	24	0.9913	59.00
	5	7.98114240E+08	29.5720	30	0.9857	55.00
	6	4.23000547E+10	35.2999	36	0.9806	53.00
	7	1.98810257E+12	40.8545	42	0.9727	47.00
	8	8.81392140E+13	46.3248	48	0.9651	44.33
	9	3.78998620E+15	51.7511	54	0.9584	43.00
	10	1.55389434E+17	57.1087	60	0.9518	41.00
	11	5.74940907E+18	62.3181	66	0.9442	37.00
	12	1.78231681E+20	67.2723	72	0.9343	31.00
	13	5.16871875E+21	72.1303	78	0.9247	29.00
	14	1.18880531E+23	76.6539	84	0.9125	23.00
	15	2.02096900E+24	80.7413	90	0.8971	17.00
	16	2.62725974E+25	84.4418	96	0.8796	13.00
7	1	1.28000000E+02	7.0000	7	1.0000	
	2	1.62560000E+04	13.9887	14	0.9992	127.00
	3	2.03200000E+06	20.9545	21	0.9978	125.00
	4	2.49936000E+08	27.8970	28	0.9963	123.00
	5	3.02422560E+10	34.8158	35	0.9947	121.00

	6	3.59882846E+12	41.7107	42	0.9931	119.00
	7	4.06667616E+14	48.5308	49	0.9904	113.00
	8	4.43267702E+16	55.2990	56	0.9875	109.00
	9	4.74296441E+18	62.0405	63	0.9848	107.00
	10	4.88525334E+20	68.7270	70	0.9818	103.00
	11	4.93410588E+22	75.3852	77	0.9790	101.00
	12	4.78608270E+24	81.9851	84	0.9760	97.00
	13	4.25961360E+26	88.4609	91	0.9721	89.00
	14	3.53547929E+28	94.8359	98	0.9677	83.00
	15	2.79302864E+30	101.1397	105	0.9632	79.00
	16	2.03891091E+32	107.3295	112	0.9583	73.00
	17	1.44762674E+34	113.4792	119	0.9536	71.00
	18	9.69909918E+35	119.5453	126	0.9488	67.00
	19	5.91645050E+37	125.4761	133	0.9434	61.00
	20	3.49070580E+39	131.3587	140	0.9383	59.00
	21	1.85007407E+41	137.0866	147	0.9326	53.00
	22	8.69534814E+42	142.6412	154	0.9262	47.00
	23	3.73899970E+44	148.0675	161	0.9197	43.00
	24	1.45820988E+46	153.3529	168	0.9128	39.00
	25	5.39537657E+47	158.5623	175	0.9061	37.00
	26	1.67256674E+49	163.5165	182	0.8984	31.00
	27	4.85044353E+50	168.3745	189	0.8909	29.00
	28	1.11560201E+52	172.8981	196	0.8821	23.00
	29	2.11964382E+53	177.1460	203	0.8726	19.00
8	1	2.56000000E+02	8.0000	8	1.0000	
	2	6.52800000E+04	15.9944	16	0.9996	255.00
	3	1.65158400E+07	23.9773	24	0.9991	253.00
	4	4.14547584E+09	31.9489	32	0.9984	251.00
	5	1.02393253E+12	39.8973	40	0.9974	247.00
	6	2.46767740E+14	47.8101	48	0.9960	241.00
	7	5.90355529E+16	55.7124	56	0.9949	239.24
	8	1.41094972E+19	63.6133	64	0.9940	239.00
	9	3.28751284E+21	71.4775	72	0.9927	233.00
	10	7.52840440E+23	79.3167	80	0.9915	229.00
	11	1.70894780E+26	87.1432	88	0.9903	227.00
	12	3.81095359E+28	94.9441	96	0.9890	223.00
	13	8.04111207E+30	102.6652	104	0.9872	211.00
	14	1.67370004E+33	110.3667	112	0.9854	208.14
	15	3.33066308E+35	118.0033	120	0.9834	199.00
	16	6.56140627E+37	125.6253	128	0.9814	197.00
	17	1.26635141E+40	133.2178	136	0.9795	193.00
	18	2.41873119E+42	140.7952	144	0.9777	191.00
	19	4.37790346E+44	148.2951	152	0.9756	181.00
	20	7.83644720E+46	155.7789	160	0.9736	179.00
	21	1.35570536E+49	163.2135	168	0.9715	173.00
	22	2.26402796E+51	170.5972	176	0.9693	167.00
	23	3.69036557E+53	177.9460	184	0.9671	163.00
	24	5.79387395E+55	185.2406	192	0.9648	157.00
	25	8.74874967E+57	192.4790	200	0.9624	151.00
	26	1.30356370E+60	199.6981	208	0.9601	149.00
	27	1.81195354E+62	206.8171	216	0.9575	139.00
	28	2.48237635E+64	213.9151	224	0.9550	137.00
	29	3.25191302E+66	220.9485	232	0.9524	131.00

30	4.12992954E+68	227.9372	240	0.9497	127.00
31	4.66682038E+70	234.7574	248	0.9466	113.00
32	5.08683422E+72	241.5256	256	0.9435	109.00
33	5.44291260E+74	248.2671	264	0.9404	107.00
34	5.60619999E+76	254.9536	272	0.9373	103.00
35	5.66226199E+78	261.6118	280	0.9343	101.00
36	5.49239413E+80	268.2117	288	0.9313	97.00
37	4.77044208E+82	274.6522	296	0.9279	86.86
38	1.39678594E+84	279.5241	304	0.9195	29.28
39	1.01965374E+86	285.7139	312	0.9157	73.00
40	7.23954154E+87	291.8636	320	0.9121	71.00
41	4.85049283E+89	297.9297	328	0.9083	67.00
42	2.95880063E+91	303.8605	336	0.9043	61.00
43	1.74569237E+93	309.7431	344	0.9004	59.00
44	9.25216956E+94	315.4710	352	0.8962	53.00
45	4.34851969E+96	321.0256	360	0.8917	47.00
46	1.86986347E+98	326.4519	368	0.8871	43.00
47	7.66644022E+99	331.8094	376	0.8825	41.00
48	2.83658288E+101	337.0189	384	0.8777	37.00
49	8.22609036E+102	341.8769	392	0.8721	29.00
50	6.66313319E+104	348.2167	400	0.8705	81.00

Appendix 2 -- Design Aid Code

Final Design Aid Program

```
/*************************************************************/
/* This program is an attempt at creating an RNS design aid.  */
/* It combines the heuristic moduli selection algorithm with  */
/* the architecture design data from section 4 of the thesis. */
/* Written by Kurt A. Locher  January 6, 1989    Lightspeed C */
/* Debugged by Kurt A. Locher until January 7, 1989  1:08am   */
/*************************************************************/

#include <stdio.h>
#include <math.h>


typedef char string[80];


/* maxmoduli indicates the maximum number of relatively prime moduli */
/* that can selected from the set of numbers less than or equal to   */
/* 2^b.  b is the index of maxmoduli, which starts at 0.             */
int maxmoduli[] = {0, 0, 0, 4, 6, 11, 16, 29, 50, 80};


main()
{

      typedef struct {
            int type;
            int bits;
            double size;
            double delay;
            char *next;
      } rec;


      int inputbits;       /* number of bits in the input */
      int coeffbits;       /* number of bits in the coeffcients */
      int b;               /* number of bits the binary channels */
      int i;
      int r_estimate;      /* initial guess at # of moduli needed */
      int r[10];           /* number of moduli needed for different b*/
      int firstb;    /* lowest bitwidth with the required dynamic range */

      int *moduli;
      int *findmoduli();

      double filterlength;       /* length of the filter in taps */
      double drange;             /* dynamic range in bits of moduli sets */
      double product;
```

```
double  totaldrange;        /* total dynamic range needed (in bits) */

FILE *nsd;                  /*  fp  for  newalgorithm_size_data  */
FILE *osd;                  /*  fp  for  oldalgorithm_size_data  */
FILE *ntd;                  /*  fp  for  newalgorithm_trans_data  */
FILE *otd;                  /*  fp  for  oldalgorithm_trans_data  */
FILE *nld;                  /*  fp  for  newalgorithm_latency_data  */
FILE *old;                  /*  fp  for  oldalgorithm_latency_data  */
FILE  *fopen();

string  otype[6];     /* text description of old design type */
double  osize[6][9];  /* size of old designs for different bitwidths */
double  odelay[6][9];       /* latency of old designs for different b */

string  ntype[6];           /* text description of new design type */
double  nsize[6][9];  /* size of new designs for different bitwidths */
double  ndelay[6][9];       /* latency of new designs for different b */

string  t[10];
string  dummy;
int j;
long int atol();
double  atof();


int  design;
char  addflag;
char  tempflag;
char  *malloc();

rec  *start;
rec  *pointer;
rec  *lastpointer;


/* First, find the number of moduli needed for each value of b */
/* from 3 to 9 */

printf("Enter number of bits in the input : ");
scanf("%d",  &inputbits);
printf("Enter number of bits in the coefficients   : ");
scanf("%d",  &coeffbits);
printf("Enter length of the filter (# of taps)   : ");
scanf("%lf",  &filterlength);

totaldrange  =  (double)inputbits  +  (double)coeffbits
                                   +   (log(filterlength)/log(2));

for(b=3; b<=9; b++)  {              /* for each value of b */
      r_estimate  =  (int)(totaldrange  /  b)  -  1;
      do {
            r_estimate++;
            if  (r_estimate  >  maxmoduli[b])  {
                  r_estimate  =  0;
```

```
                        firstb = b;
                        break;      /* do loop */
                    }
                    moduli = findmoduli(r_estimate, b, &product);
                    drange = log(product)/log(2);
                } while (drange < totaldrange);

                r[b] = r_estimate;

/ *             if (r[b] != 0) {
                    printf("\nThe optimal moduli set for b = %d is ...", b);
                    for(i=0; i<r[b]; i++) {
                        printf("%d\t", moduli[i]);
                    }
                    printf("\nThe product is %.8e for %lf bits of
precision\n\n",
                        product, (log(product)/log(2)));
                }
* /
        }
        firstb++;

        /* Now, load the size and latency data from files */

        if((osd = fopen("oldalgorithm_size_data", "r")) == NULL)
            printf("Error opening oldalgorithm_size_data\n");
        for(i=0; i<4; i++) {
            fscanf(osd, "%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s", otype[i],
                t[3], t[4], t[5], t[6], t[7], t[8], t[9] );

            for (j=3; j<=9; j++)
                osize[i][j] = r[j] * atof(t[j]);
        }
        fclose(osd);

        if ((old = fopen("oldalgorithm_latency_data", "r")) == NULL)
            printf("Error opening oldalgorithm_latency_data\n");
        for(i=0; i<4; i++) {
            fscanf(old, "%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s", dummy,
                t[3], t[4], t[5], t[6], t[7], t[8], t[9] );

            for (j=3; j<=9; j++)
                odelay[i][j] = atof(t[j]);
        }
        fclose(old);


        /* Now lets prune the number of alternatives a bit */

        start = (rec *)malloc(sizeof(rec));
        start->size = 9e30;
        start->delay = 9e30;
        start->next = NULL;
        for(design=0; design<4; design++) {
```

```
            for(b=firstb; b<=9; b++) {
                  pointer = start;
                  lastpointer = NULL;
                  addflag = 1;
                  do {
                        tempflag = 0;
                        if (pointer->size > osize[design][b])       tempflag++;
                        if (pointer->delay > odelay[design][b])  tempflag++;

                        if (tempflag == 0)   addflag = 0;
                        if (tempflag == 2) {
                              /* delete current record */
                              if (lastpointer == NULL) { /* if first in list */
                                          start = (rec *)pointer->next;
                                          free(pointer);
                                          pointer = start;
                              }
                              else {
                                    lastpointer->next = pointer->next;
                                    free(pointer);
                                    pointer = (rec *)pointer->next;
                              }
                        }
                        else {
                              /* otherwise, just step to the next record */
                              lastpointer = pointer;
                              pointer = (rec *)pointer->next;
                        }
                  } while (pointer != NULL);

                  if (addflag == 1) {
                        pointer = (rec *)malloc(sizeof(rec));
                        pointer->type = design;
                        pointer->bits = b;
                        pointer->size = osize[design][b];
                        pointer->delay = odelay[design][b];
                        pointer->next = NULL;
                        if (start == NULL)
                              start = pointer;
                        else
                              lastpointer->next = (char *)pointer;
                  }
            }
      }

      /* And print the results */

      pointer = start;
      printf("\n\nThe old designs which provide the best size/speed
combinations are\n");
      do {
            printf("%s design with %d  %d bit moduli --> size %.0f,  delay
%.2f\n",
              otype[pointer->type], r[pointer->bits], pointer->bits, pointer->size,
```

```
            pointer->delay);
        pointer = (rec *)pointer->next;
    } while (pointer != NULL);


/* And now for something a little different */

    if((nsd = fopen("newalgorithm_size_data", "r")) == NULL)
        printf("Error opening newalgorithm_size_data\n");
    for(i=0; i<3; i++) {
        fscanf(nsd, "%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s", ntype[i],
                    t[3], t[4], t[5], t[6], t[7], t[8], t[9] );

        for (j=3; j<=9; j++)
            nsize[i][j] = r[j] * atof(t[j]);
    }
    fclose(nsd);

    if ((nld = fopen("newalgorithm_latency_data", "r")) == NULL)
        printf("Error opening newalgorithm_latency_data\n");
    for(i=0; i<3; i++) {
        fscanf(nld, "%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s", dummy,
                    t[3], t[4], t[5], t[6], t[7], t[8], t[9] );

        for (j=3; j<=9; j++)
            ndelay[i][j] = atof(t[j]);
    }
    fclose(nld);

    /* Now lets prune the number of alternatives a bit */

    start = (rec *)malloc(sizeof(rec));
    start->size = 9e30;
    start->delay = 9e30;
    start->next = NULL;
    for(design=0; design<3; design++) {
        for(b=firstb; b<=9; b++) {
            pointer = start;
            lastpointer = NULL;
            addflag = 1;
            do {
                tempflag = 0;
                if (pointer->size > nsize[design][b])       tempflag++;
                if (pointer->delay > ndelay[design][b]) tempflag++;

                if (tempflag == 0)  addflag = 0;
                if (tempflag == 2) {
                    /* delete current record */
                    if (lastpointer == NULL) { /* if first in list */
                                start = (rec *)pointer->next;
                                free(pointer);
                                pointer = start;
                    }
```

```
                        else {
                                lastpointer->next = pointer->next;
                                free(pointer);
                                pointer = (rec *)pointer->next;
                        }
                }
                else {
                        /* otherwise, just step to the next record */
                        lastpointer = pointer;
                        pointer = (rec *)pointer->next;
                }
        } while (pointer != NULL);

        if (addflag == 1) {
                pointer = (rec *)malloc(sizeof(rec));
                pointer->type = design;
                pointer->bits = b;
                pointer->size = nsize[design][b];
                pointer->delay = ndelay[design][b];
                pointer->next = NULL;
                if (start == NULL)
                        start = pointer;
                else
                        lastpointer->next = (char *)pointer;
        }
    }
  }

  /* And print the results */

  pointer = start;
  printf("\n\nThe new designs which provide the best size/speed
combinations are\n");
  do {
        printf("%s design with %d  %d bit moduli --> size %.0f, delay
%.2f\n",
          ntype[pointer->type], r[pointer->bits], pointer->bits, pointer-
>size,
          pointer->delay);
        pointer = (rec *)pointer->next;
  } while (pointer != NULL);
}


/*********************************************************/
/* Function findmoduli returns the "optimal" set of moduli with the      */
/* desired number of bits and elements.  It operates by recursively      */
/* calling a sub-optimal moduli selection function with a list of        */
/* excluded numbers.                                                     */
/* Inputs:  # of moduli, # of bits/moduli                                */
/* Outputs: moduli set, log2 (product of moduli set)                     */
/*********************************************************/
int *findmoduli(number, bits, product)
int number;             /* number of moduli in set */
```

```
int bits;                   /* maximum number of bits per modulus */
double *product;   /* product of best moduli set, returned */
{

        char  doitagain;    /* done flag */
        char  *malloc();    /* dynamic memory allocation */
        char  *realloc();   /* dynamic memory allocation */

        int i, j;           /* looping variables */
        int  numexc;        /* size of exclusion set passed */
        int  *exclude;      /* exclusion set */
        int  *result;  /* best set of moduli */
        int  *mod;          /* current set of moduli */
        int  power(); /* integer power function */

        double  temp;       /* product of current moduli set */


        /* Initialize stuff... */
        mod = (int *)malloc(number * sizeof(int));
        result = (int *)malloc(number * sizeof(int));
        exclude = (int *)malloc(sizeof(int));

        numexc = 0;                     /* start with no exclusions */
        mod[0] = power(2, bits);    /* always include highest power of 2 */

        /* Take first stab with sub-optimal algorithm */
        findset(mod, number, exclude, numexc);
        *product = mod[0];
        for(i=1; i<number; i++)
                *product = *product * mod[i];

        /* recursively call suboptimal algorithm successively excluding */
        /* each member of the current best moduli set */
        do {
                doitagain = 0;
                for(i=0; i<number; i++)
                        result[i] = mod[i];
                numexc++;
                exclude = (int *)realloc(exclude, numexc*sizeof(int));
                for(i=1; i<number; i++) {
                        exclude[numexc - 1] = result[i];
                        findset(mod, number, exclude, numexc);
                        temp = mod[0];
                        for(j=1; j<number; j++)
                                temp = temp * mod[j];

                        /* Is the current moduli set better than the best */
                        if (temp > *product) {
                                *product = temp;
                                doitagain = 1;
                                break;              /* break out of the for loop */
                        } /* if */
                } /* for */
```

```
        } while(doitagain);

        return(result);
}


/*********************************************************/
/* Function findset executes the basic flawed algorithm for finding     */
/* a moduli set with the addition of select number exclusion.  The       */
/* basic algorithm starts with the highest possible moduli in a b        */
/* binary representation (2 to the power of b).  It then counts down     */
/* the odd numbers less than this highest modulus including those        */
/* that are relatively prime to the currently existing set.  By          */
/* the exclusion feature allows a calling function to compensate for     */
/* the the flaw in this algorithm by excluding multiple factor           */
/* numbers.                                                              */
/*********************************************************/
int findset(modset, lenset, exclude, lenexc)
int *modset; /* predimensioned array to hold moduli, modset[0] = 2^n */
int lenset;          /* length of requested moduli set */
int *exclude;        /* predimensioned and initialized exclude set */
int lenexc;          /* length of exclude set */
{

        char success;        /* flag */

        int i, j;            /* looping variables */
        int newmod;          /* potential new member of moduli set */
        int gcd();           /* greatest common denominator function */

        /* find first odd number that does not conflict with exclude set */
        newmod = modset[0] - 1;
        if (lenexc != 0) {
                do {
                        success = 1;
                        for(j=0; j<lenexc; j++) {
                                if (newmod == exclude[j]) {
                                        newmod -= 2;        /* go to next greatest odd
* /
                                        success = 0;
                                        break; /* the for loop */
                                }
                        }
                } while(!success);
        }
        modset[1] = newmod;

        /* Using first odd element as a seed, find the remaining relatively */
        /* prime elements to complete the set.  For these elements not only */
        /* must the exclude list be checked, but also they must be           */
        /* relatively prime to the existing elements */
        for(i=2; i<lenset; i++) {
                newmod = modset[i-1] - 2;
```

```
            do {
                  /* check exclude list */
                  if (lenexc != 0) {
                        do {
                              success = 1;
                              for(j=0; j<lenexc; j++) {
                                    if (newmod == exclude[j]) {
                                          newmod -= 2;     /* go to next
greatest odd */
                                          success = 0;
                                          break; /* the for loop */
                                    }
                              }
                        } while(!success);
                  }

                  /* check for relative primality */
                  success = 1;
                  for(j=1; j<i; j++) {
                        if (gcd(modset[j], newmod) != 1) {
                              newmod -= 2;       /* go to next greatest odd
* /
                              success = 0;
                              break; /* the for loop */
                        }
                  }

            } while(!success);

            modset[i] = newmod;

      } /* for */
}


int power(num, pow)
int num;
int pow;
{
      int i;
      int res;

      res = num;
      for(i=1; i<pow; i++)
            res = num * res;
      return(res);
}


/*********************************************************/
/* function gcd returns the greatest commmon divisor of two integers   * /
/* using Euclid's Algorithm...                                         * /
```

```
/*********************************************************/
int gcd(a, b)
int a;
int b;
{
        int temp;
        int r;

        /* make sure a > b */
        if (a < b) {
                temp = a;
                a = b;
                b = temp;
        }

        /* Euclid's Algorithm */
        do {
                r = a % b;
                a = b;
                b = r;
        } while(r > 0);

        return(a);
}
```

Moduli Selection Algorithm

```
/*********************************************************/
/* This program is an attempt at creating an RNS design aid.    */
/* It uses a recursive method that is based on some of the       */
/* basic characteristics of an optimal moduli set.               */
/* Written by Kurt A. Locher  August 29, 1988                    */
/* Ported to the Macintosh    November 12, 1988                  */
/*********************************************************/

#include <stdio.h>
#include <math.h>


main()
{
        int  nummod;
        int  numbits;
        int i;
        int  *moduli;
        int  *findmoduli();

        double  product;

        do {
                printf("Enter bitlength of moduli : ");
                scanf("%d",  &numbits);
                printf("Enter number of moduli   : ");
                scanf("%d",  &nummod);
                moduli = findmoduli(nummod, numbits);
                product = 1;
                printf("\nThe optimal moduli set is ...");
                for(i=0; i<nummod; i++) {
                        product = product * moduli[i];
                        printf("%d\t",  moduli[i]);
                }
                printf("\nThe product is %.8e for %lf bits of precision\n\n",
                                product,  (log(product)/log(2)));
        } while(numbits > 2);
}


/*********************************************************/
/* Function findmoduli returns the "optimal" set of moduli with the  */
/* desired number of bits and elements.  It operates by recursively  */
/* calling a sub-optimal moduli selection function with a list of    */
/* excluded numbers.                                                 */
/*********************************************************/
int  *findmoduli(number,  bits)
int  number;        /* number of moduli in set */
int bits;           /* maximum number of bits per modulus */
{
```

```
char  doitagain;     /* done flag */
char  *malloc();     /* dynamic memory allocation */
char  *realloc();    /* dynamic memory allocation */

int i, j;            /* looping variables */
int  numexc;         /* size of exclusion set passed */
int  *exclude;       /* exclusion set */
int  *mod;           /* current set of moduli */
int  *result;  /* best set of moduli */
int  power(); /* integer power function */

double  product;     /* product of best moduli set */
double  temp;        /* product of current moduli set */


/* Initialize stuff... */
mod = (int *)malloc(number * sizeof(int));
result = (int *)malloc(number * sizeof(int));
exclude = (int *)malloc(sizeof(int));

numexc = 0;                          /* start with no exclusions */
mod[0] = power(2, bits);    /* always include highest power of 2 */

/* Take first stab with sub-optimal algorithm */
findset(mod, number, exclude, numexc);
product = mod[0];
for(i=1; i<number; i++)
      product = product * mod[i];

/* recursively call suboptimal algorithm successively excluding */
/* each member of the current best moduli set */
do {
      doitagain = 0;
      for(i=0; i<number; i++)
            result[i] = mod[i];
      numexc++;
      exclude = (int *)realloc(exclude, numexc*sizeof(int));
      for(i=1; i<number; i++) {
            exclude[numexc - 1] = result[i];
            findset(mod, number, exclude, numexc);
            temp = mod[0];
            for(j=1; j<number; j++)
                  temp = temp * mod[j];

            /* Is the current moduli set better than the best */
            if (temp > product) {
                  product = temp;
                  doitagain = 1;
                  break;              /* break out of the for loop */
            } /* if */
      } /* for */
}  while(doitagain);

return(result);
```

```
}


/*********************************************************/
/* Function findset executes the basic flawed algorithm for finding      */
/* a moduli set with the addition of select number exclusion.  The        */
/* basic algorithm starts with the highest possible moduli in a b         */
/* binary representation (2 to the power of b).  It then counts down      */
/* the odd numbers less than this highest modulus including those         */
/* that are relatively prime to the currently existing set.  By           */
/* the exclusion feature allows a calling function to compensate for      */
/* the the flaw in this algorithm by excluding multiple factor            */
/* numbers.                                                               */
/*********************************************************/
int findset(modset, lenset, exclude, lenexc)
int *modset; /* predimensioned array to hold moduli, modset[0] = 2^n */
int lenset;          /* length of requested moduli set */
int *exclude;        /* predimensioned and initialized exclude set */
int lenexc;          /* length of exclude set */
{

      char success;        /* flag */

      int i, j;            /* looping variables */
      int newmod;          /* potential new member of moduli set */
      int gcd();           /* greatest common denominator function */

      /* find first odd number that does not conflict with exclude set */
      newmod = modset[0] - 1;
      if (lenexc != 0) {
            do {
                  success = 1;
                  for(j=0; j<lenexc; j++) {
                        if (newmod == exclude[j]) {
                              newmod -= 2;       /* go to next greatest odd
* /
                              success = 0;
                              break; /* the for loop */
                        }
                  }
            } while(!success);
      }
      modset[1] = newmod;

      /* Using first odd element as a seed, find the remaining relatively */
      /* prime elements to complete the set.  For these elements not only */
      /* must the exclude list be checked, but also they must be          */
      /* relatively prime to the existing elements */
      for(i=2; i<lenset; i++) {
            newmod = modset[i-1] - 2;

            do {
                  /* check exclude list */
```

```
                if (lenexc != 0) {
                        do {
                                success = 1;
                                for(j=0; j<lenexc; j++) {
                                        if (newmod == exclude[j]) {
                                                newmod -= 2;        /* go to next
greatest odd */
                                                success = 0;
                                                break; /* the for loop */
                                        }
                                }
                        } while(!success);
                }

                /* check for relative primality */
                success = 1;
                for(j=1; j<i; j++) {
                        if (gcd(modset[j], newmod) != 1) {
                                newmod -= 2;        /* go to next greatest odd
* /
                                success = 0;
                                break; /* the for loop */
                        }
                }

        } while(!success);

        modset[i] = newmod;

    } /* for */
}



int power(num, pow)
int num;
int pow;
{
    int i;
    int res;

    res = num;
    for(i=1; i<pow; i++)
        res = num * res;
    return(res);
}


/*********************************************************/
/* function gcd returns the greatest commmon divisor of two integers   * /
/* using Euclid's Algorithm...                                         * /
/*********************************************************/
int gcd(a, b)
int a;
```

```
int  b;
{
        int  temp;
        int  r;

        /* make sure a > b */
        if (a < b) {
                temp = a;
                a = b;
                b = temp;
        }

        /* Euclid's Algorithm */
        do {
                r = a % b;
                a = b;
                b = r;
        } while(r > 0);

        return(a);
}
```

References

General

Herstein, I.N. *Abstract Algebra*, MacMillan Publishing, New York, 1986

Knuth, D.E. *The Art of Computer Programming*, Vol 2 Seminumerical Algorithms, Addison-Wesley, Reading, MA, 1981 (pp 178-197 & 268-277)

McClellen, J.H. & Rader, C.M. *Number Theory in Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1979

Oppenheim, A.V. & Shafer, R.W. *Digital Signal Processing*, Prentice-Hall, New York, 1975

Rabiner, L.R. & Gold, B. Theory and Application of Digital Signal Processing, Prentice-Hall, Englewood Cliffs, NJ, 1975

Szabo, N.S. & Tanaka, R.I. *Residue Arithmetic and its Application to Computer Technology*, McGraw-Hill, New York, 1967

Taylor, F.J. "Residue Arithmetic: A Tutorial with Examples", IEEE Computer, May 1984

Architecture

Bayoumi, M.A., et al. "A VLSI Implementation of Residue Adders", IEEE Transactions on Circuits and Systems, Vol CAS-34, No 3, March 1987

Chaing, C-L "Residue Arithmetic and VLSI", IEEE 1983

Huang, C.H. "Implementation of a Fast Digital Processor Using the Residue Number System", IEEE Transactions on Circuits and Systems, Vol CAS-28, January 1981

Jenkins, W.K., et al, "The Use of Residue Number Systems in the Design of Finite Impulse Response Digital Filters", IEEE Transactions on Circuits and Systems, April 1977

Johnson, B.L., et al, "The Residue Number System for VLSI Signal Processing", SPIE Vol 696 Advanced Algorithms and Architectures for Signal Processing, 1986

Jullien, G.A. "Residue Number Scaling and Other Operations using ROM Arrays", IEEE Transactions on Computers, C-27, No 4, April 1978

Key, E.L. "Digital Signal Processing with Residue Number Systems", IEEE 1983

Kung, H.T. "Why Systolic Architectures?", IEEE Computer, Jan 1982

Musicus, B.R. "Architectures for Signal Processing", MIT Course Handout

Soderstrand, M.A. "A New Hardware Implementation of Modulo Adders for Residue Number Systems"

Taylor, F.J., et al. "An Efficient Residue-to-Decimal Converter", IEEE Transactions on Circuits and Systems, Vol CAS-28, No 12, December 1981

Taylor, F.J. "A VLSI Residue Arithmetic Multiplier", IEEE Transactions on Computers, Vol C-31, No 6, June 1982

ZORAN Corporation, "DFP Family Overview" and "ZR33881 Digital Filter Processor Data Sheet"

Complex Residue Arithmetic

Baranieka, A.Z., et al, "Residue Number System Implementations of Number Theoretic Transforms in Complex Residue Rings", IEEE Transactions on ASSP, Vol ASSP-28, No 3, June 1980

Jenkins, W.K. "Quadratic Modular Number Codes for Complex Digital Signal Processing", IEEE ISCAS 1984

Jullien, G.A., et al, "Complex Digital Signal Processing over Finite Rings", IEEE Transactions on Circuits and Systems, Vol CAS-34, No 4, April 1987

Krishnan, R., et al, "The Modified Quadratic Number System (MQRNS) for Complex High-Speed Signal Processing", IEEE Trasactions on Circuits and Systems, Vol CAS-33, No 3, March 1986

Krishnan, R., et al, "Complex Digital Signal Processing Using Quadratic Number System", IEEE Trasactions on ASSP, Vol ASSP-34, No 1, February 1986

Soderstrand, M.A. "Applications of Quadratic-Like Complex Residue Number System Arithmetic to Ultrasonics", IEEE International Conference ASSP 1984, vol 2, March 1984